# Building a bridge between Goal-Oriented Requirements with KAOS and event-B System Specifications

Devroey, Xavier

*Publication date:*
2010

*Document Version*
Peer reviewed version


**Facultés Universitaires Notre-Dame de la Paix, Namur**
**Faculté d'Informatique**

**Année académique 2009-2010**

---

# Building a bridge between Goal-Oriented Requirements with KAOS and Event-B System Specifications

*Xavier Devroey*

Promoteur : **Wim Vanhoof**
Maître de stage : **Christophe Ponsard**

Mémoire présenté en vue de l'obtention du grade de master en sciences informatiques

# Résumé

Ce mémoire présente des techniques pour lier des modèles d'exigences orientés buts (GORE) et des spécifications formelles exprimées en termes plus opérationnel. Plus particulièrement, l'objectif était de produire une méthode permettant de dériver un modèle Event-B à partir d'un modèle KAOS qui s'appuie au maximum sur les possibilités de ces langages, qui garantisse une traçabilité fine et qui soit la plus automatisée possible.

Après l'analyse d'un certain nombre d'approches existantes, aucune méthode ne semble répondre au problème. Par conséquent une approche alternative a été conçue en se concentrant principalement sur la mise en correspondance d'agents GORE et de machines Event-B.

Le travail repose entièrement sur UML-B pour la dérivation des modèles de données. Des extensions récentes d'Event-B traitant de la décomposition de machine sont aussi utilisées. Une machine initiale correspondant au modèle de données KAOS est d'abord créée. Cette machine est l'équivalent du système entier, capable de controler toutes les données. La machine initiale est ensuite décomposée en machines agent, exprimées de manière plus fine en se basant sur leurs capacités de contrôler certaines données. Enfin, le comportement des machines agent est raffiné pour correspondre au comportement déclaré dans le modèle KAOS.

L'approche a été partiellement implémentée dans un prototype qui utilise des technologies de transformation de modèle à modèle (EMF - ATL), et validée sur différents cas.

## Mots clés

Event-B, KAOS, ingénierie des exigences, ingénierie dirigée par les modèles, méthodes formelles, méthodes orientées buts

# Abstract

This master thesis presents techniques for connecting requirements models expressed in a goal-oriented requirements engineering (GORE) paradigm into more operational specifications expressed in Event-B. More specifically, the objective was to produce a method that derives an Event-B model from a KAOS model, that relies on the semantics of those two languages, that guarantee a fine-grained traceability and that is as automatic as possible.

After reviewing a number of existing approaches, none of those methods seem to answer the problem. Consequently an alternative approach was designed with the central focus of mapping GORE agents to Event-B machines.

The work fully relies on the UML-B work for mapping data models. Recent Event-B extensions about machine decomposition are also used to decompose an initial system level machine into more finer grained agent machines based on their ability to control a specific piece of information. Finally the agent machines are refined to match the behaviour declared in the KAOS model.

The approach has been partially implemented in a prototype that uses model to model transformation technologies (EMF - ATL), and has been validated on different cases.

## Keywords

Event-B, KAOS, requirements engineering, model driven engineering, formal methods, goal orientation

# Avant-propos

Je remercie toutes les personnes qui ont participé directement ou indirectement à la réalisation de ce mémoire :

En particulier Wim Vanhoof, professeur à la faculté d'Informatique des Facultés Universitaires Notre-Dame de la Paix de Namur et promoteur de ce mémoire, qui s'est toujours montré à l'écoute et très disponible, pour l'aide et le temps qu'il a bien voulu me consacrer.

Christophe Ponsard, responsable du département Software & System Engineering au CETIC et responsable de mon stage, pour son aide tout au long de la rédaction de ce mémoire, ses conseils et son expérience.

Le personnel du CETIC pour leur accueil et les professeurs des Facultés Universitaires Notre-Dame de la Paix de Namur pour le savoir qu'ils m'ont transmis durant ces trois années.

Enfin, j'adresse mes plus sincères remerciements à mes parents, pour leur soutien inconditionnel et la patience dont ils ont fait preuve et à tous mes proches et amis, qui m'ont soutenu et encouragé au cours de la réalisation de ce mémoire.

# Contents

# Chapter 1

# Introduction

Application development is classically defined as a process decomposed in several phases. The first one is the *requirements engineering phase* which aims at capturing the requests of the client and writing them down in a requirements document. In the second phase, the *specification phase*, a system that responds to the requirements of the client will be imagined and described in a specifications document. The third phase is the *design phase* where the specifications are refined to get a precise architecture of the future application. The last phase is the *coding phase*, where the architecture is implemented using a programming language.

As the chosen programming language is the formalism used for the coding phase, the other phases will have their own language, *e.g.* class diagrams and sequence diagrams for the design phase. Those languages may be less or more formal according to their purposes and stakeholders, *i.e.* people involved in the application development, that will read and use them. For instance, natural language may be used for the requirements document and a purely mathematically based language may be used for the specifications of the system. All languages have their advantages and disadvantages, natural language may be easily understood by the client that will validate the requirements document, but it may be ambiguous for the analyst that will have to establish specifications based on this document. On the contrary, mathematically based languages are unambiguous, but are hard to understand for non-specialists.

Another major side of a development process are the tests taking place at different moments in the process. Tests will generally occur at the end of a phase, *e.g.* testing the code to ensure that it is bug-free or testing that the different elements described in the design have effectively been implemented. According to the used language, the tests will be more or less automated. One of the problems usually encountered with tests in a development process is the test's coverage, especially with the source code which is usually too big to be completely tested. This may be problematic, in the case of a critical

system where a failure in the system may have important consequences, going from recovery issues to money or even human lives loss.

One way to overcome this is to use formal methods in the specification and/or the design phases. As we will see in chapter 3, a formal method uses a formal language composed of syntax with a semantic relying on a mathematical substratum and a proof theory defining rules for inferring useful information. For instance, if we use a formal method to establish the specifications, it will be possible to prove that the specifications are correct, *i.e.* the specifications are consistent and correct with themselves, but without taking care whether the specifications meets the requirements or not. If we want to prove such a thing, the requirements have to be written in a formal language compatible with the language used to express the specifications. The reader could easily understand here that using formal specifications may be quite heavy, as well for the analyst which will have to write down the specifications, as for the client, usually non-expert in the used formal language, that will have to validate them.

To avoid such a situation, the idea is to have an uninterrupted chain between the requirements, expressed in such a way that they are easily understood by the client, and the source code. In this case, the specifications would be formally derived from the requirements, the design is formally derived from the specifications and the code is automatically generated from the design. Such derivation chains already exist partially. In *Event-B*, a formal method used for the specification phase and based on a refinement strategy, a general model is refined to be made more precise with a proof at each refinement that the concrete model does not contradict the abstract one. When the model is precise enough, a B model based on the *B-Method*, also sometimes called Classical-B, may be automatically generated. As for Event-B, Classical-B is based on a refinement strategy where the model is made more concrete at each step. When the B model is precise enough, the source code may be automatically generated. Contrary to handwritten source code, this generated code has been proved equivalent to the specification expressed in the general Event-B model and is thus bug-free, although it may contain what we may call business errors. Business errors are coming from a "modelisation error", *e.g.* misunderstanding of the requirements expressed by the client.

In this case we have an uninterrupted chain between the specifications and the source code, but there is still **a gap between requirements and specifications**. As the requirements have to be formalized in a way or another, we chose here to use **KAOS**: a goal oriented methodology. As for **Event-B**, KAOS is based on a refinement strategy. Starting from high-level goals saying why the future system has to be build, arriving at a set of *requirements* and *expectations* saying **how** the future system will fulfil those goals and **who** will be involved in this fulfilment (the implied *agents*). KAOS also include a formal layer by permitting the definition of the goals,

requirements and expectations with **linear temporal logic** formula.

This thesis is structured as follows: chapter 2 will present the main concepts that may be found in requirements engineering, more particularly goal oriented requirements engineering and KAOS. Chapter 3 will introduce the notion of formal method and will present in more details the Event-B method. In chapter 4 we design our approach to build a bridge between KAOS and Event-B that fulfils a number of objectives as none of the existing approaches (presented in chapter 5) could meet them all. A comparative discussion is also presented at end of chapter 5. Chapter 6 will conclude by summarising the problem, highlighting contribution, pointing some limits and sketching some possible further work to address them.

# Chapter 2

# Requirements Engineering

This chapter introduces general notions used in requirements engineering. The first section describes what requirements engineering is about. The second section introduces goal oriented requirements engineering, a goal driven approach to perform tasks that can be found in a requirements engineering process. The last section describes KAOS, a goal oriented requirements engineering method. Most of this chapter is inspired on van Lamsweerde's book [van Lamsweerde, 2009].

## 2.1 Overview of Requirements Engineering

Requirements engineering is concerned about the definition and the understanding of a problem. In more details, it is focused on the discovering, understanding, formulation, analysis and consensus formulations of the why, what and who dimensions of the problem. Figure 2.1 presents the links between the different dimensions in a synthetic way.

### Why-dimension

The Why-dimension wants to define why the problem needs to be solved. It is expressed in terms of goals that must be reached by the system under development. It includes the analysis of each alternative with its advantages and disadvantages and the management of conflicts between different points of view and self-interests in order to have a coherent set of goals.

### What-dimension

The What-dimension is interested in the functional services that the system-to-be should provide to satisfy the objectives defined in the why-dimension. Those services may be automated in software or may be manual procedures and generally rely on the assumptions made on the system to work correctly.

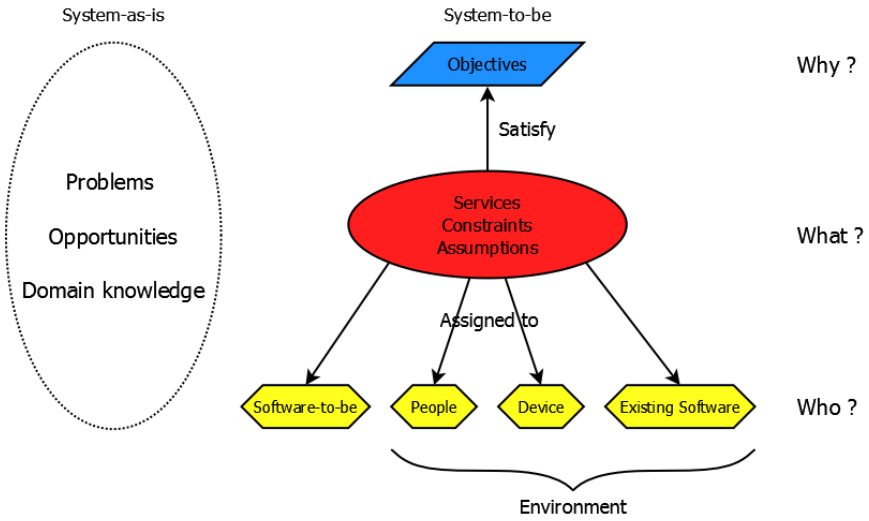


Figure 2.1: Why, what and who dimensions [van Lamsweerde, 2009]

They will have to respect some constraints about performance, security, usability, interoperability and cost.

#### Who-dimension

The Who-dimension establishes the responsibilities between the services and the human, software and hardware components of the system-to-be. When different possibilities exist for an assignment of a responsibility, the advantages and disadvantages will be evaluated reminding that different possibilities may lead to different more or less automated systems.

### 2.1.1 Problems Context

Problems generally come with a particular context larger than the problems themselves. This context may be part of a complex organizational, technical or physical world with its own rules. The main goal of a project will be to construct a machine to resolve a problem and thus improve the context.

Figure 2.2 presents a general view of a machine and its environment. The environment is part of the context on which the effects of the machine can be seen. The machine is composed of a software and hardware part, and interacts with the environment through shared phenomena. Those phenomena are monitored or controlled by the machine in order to implement the specifications [Lapouchnian, 2005, van Lamsweerde, 2009].

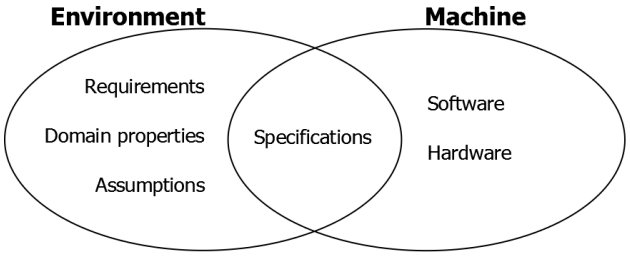


Figure 2.2: General view of a machine and its environment [van Lamsweerde, 2009]

### Definitions

Before going further, we give some definitions proposed by van Lamsweerde and Lapouchnian in [van Lamsweerde, 2009, Lapouchnian, 2005] :

A *System* is a set of components each one interacting with another in order to meet a global objective. What we will call the system-as-is is the system before the construction of the machine. The system-to-be will denote the system as it should be when the machine, also called *software-to-be*, will be implemented and working.

The notion of *descriptive statement* is defined by van Lamsweerde in [van Lamsweerde, 2009] as:

> "*Descriptive statements state properties about the system that hold regardless of how the system behaves. It holds typically because of some natural law or physical constraint*".

The notion of *prescriptive statement* is defined by van Lamsweerde in [van Lamsweerde, 2009] as:

> "*Prescriptive statements state desirable properties about the system that may hold or not depending on how the system behaves. Such statement needs to be enforced by system components*".

*Requirements*, also called system requirements, are prescriptive statements formulated in terms of environment phenomena that describe desired conditions over those phenomena. They will be enforced by the software-to-be and other system components.

*Specifications*, also called software requirements, are prescriptive statement formulated in terms of phenomena shared between the software and the environment. Those statements will be enforced exclusively by the software-to-be.

*Domain properties* are properties of the environment which are expected to always hold regardless of the system's behaviours and even regardless of whether there will be any software-to-be or not.

*Assumptions* , also called expectations, are generally prescriptive statements formulated in terms of environment phenomena that have to be satisfied by agents in the environment.

The *link* between all those elements may be formulated as follow [van Lamsweerde, 2009]:

$$Specifications, Domain\ properties, Assumptions \models Requirements$$

This can be read:

> "*If the specifications are satisfied by the software, the assumptions are satisfied by the environment, the domain properties hold and all those statements are consistent with each other, then the requirements are satisfied by the system*".

### 2.1.2 Requirements Engineering Concerns

Requirements engineering is concerned with the left set in figure 2.2, it captures and describes specifications, assumptions, domain properties and requirements. Requirements engineering does not care about the how-dimension, which states how the specifications will be implemented by the software. This dimension is part of the software-design process.

We can now have the general definition of requirements engineering give by van Lamsweerde in [van Lamsweerde, 2009]:

> "*Requirements engineering is a coordinated set of activities for exploring, evaluating, revising and adapting the objectives, capabilities, qualities, constraints and assumptions that the system-to-be should meet based on problems raised by the system-as-is and opportunities provided by new technologies*".

## 2.2 Overview of Goal Oriented Requirements Engineering

Goal Oriented Requirements Engineering (GORE) is a goal driven approach to perform tasks defined in a requirement engineering process. Contrary to requirements engineering techniques concerned with "late-phase", like use-case modelling, where initial functional requirement statements are clarified and analysed to detect ambiguities, incompleteness or inconsistencies, the GORE approach mainly address the Why-dimension of the system-to-be [van Lamsweerde, 2009, Letier, 2001].

It is focused on system objectives as a core abstraction and captures this abstraction through goals. The main concern will be the exploration of the user's goals and the analysis of the different possible systems which may satisfy those goals [van Lamsweerde, 2009,Letier, 2001]. Two complementary frameworks working with goals exist:

*NFR* is concerned with the evaluation and selection of alternatives respecting qualitative non-functional goals, *e.g.* usability, performance, accuracy, security [Chung et al., 2000].

*KAOS* is concerned with the generation of alternative systems from high level goals expressed in linear temporal logic [Letier, 2001].

In the remainder of this work, we will rather explore the KAOS point of view than NFR, *e.g.* the definition of soft-goal and non-functional goal which are identical within NFR [Chung et al., 2000], but are two distinct concepts in a KAOS context.

### 2.2.1 Goal

As said before, GORE is all about goals, but what is a goal? A goal is expressed as a prescriptive statement about an objective the system has to reach through the cooperation of its agents. Goals may be formulated at different levels of abstraction, from high level strategic concerns, to low level technical concerns.

During the elicitation process, goals will be refined into sub-goals that contribute to the realization of the parent goal. Goals may also be abstracted into a more general parent goal to which they contribute. The finer-grained a goal is, the fewer agents it will need to be satisfied. It is important to underline here that a GORE process is generally not a top-down approach, some goals are identified and from them, sub-goals and parent goals will appear from elicitation.

#### Goal Taxonomies

There exist several goal taxonomies used to facilitate elicitation, one of the most common is the distinction between functional goals and non-functional goals:

*Functional goals* express the intention hidden behind a system service. For instance, an information goal *"Notify the Accounts department that an invoice can be sent to the Client"* is a functional goal concerned with keeping the *Accounts department* agent informed about system states.

*Non functional goals* express a quality or constraint on a service or the development process. It may be about safety, security, performance, cost, *etc.* For instance *"Products have to be sent to the client maximum four days after they have been ordered"* is a time performance goal.

This distinction between functional and non-functional goals, which are goal *categories*, is a fuzzy classification used in the elicitation process to check if all aspects of the system-to-be have been considered. It must not be confused with the distinction between behavioural and soft-goals, which are goal *types*:

A *Soft-goal* prescribes preferences among alternative system behaviours. Its satisfaction cannot be established in a clear-cut sense, but a well-defined satisfaction measure criterion has to be given. Sub-types have been defined. Among the most frequently used ones: improve, increase, reduce, maximize and minimize. They correspond to different types of measure criteria.

A *behavioural goal* prescribes the desired behaviour in a declarative way. It implicitly defines the maximal set of admissible states. Sub-types corresponding to particular behaviours have been defined. Among the most frequently used: achieve, cease, maintain and avoid. Figure 2.3 shows this classification in a synthetic way.

The goal *type* classification is a semantic classification, in the sense that a goal can be satisfied by the system behaviour in a clear-cut sense or not.

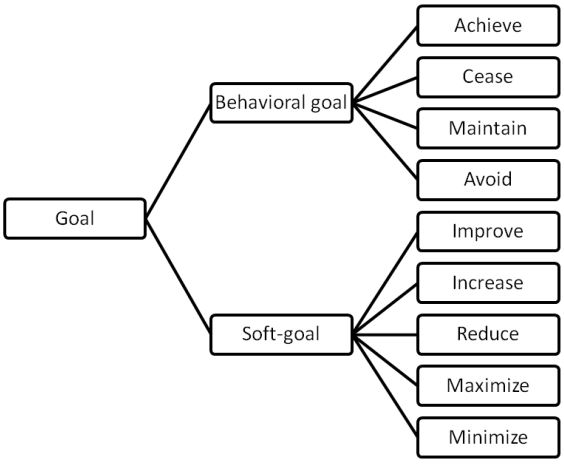


Figure 2.3: Goal type taxonomy [van Lamsweerde, 2009]

The most commonly used behavioural goal type is the Achieve one, which states that sooner-or-later a certain condition will be reached. The Cease goal type states that sooner-or-later the negation of a certain condition will be true. Maintain and Avoid types state that a certain condition will always or never be true.

Improve soft-goal types states that a certain condition should be enhanced. Increase and Reduce types are used with a quantity criterion while Maximize and Minimize are used with the most general notion of objective function. More details about soft-goal types can be found in [van Lamsweerde, 2009].

### 2.2.2 Agent

Agents are active components of the system playing a role to satisfy some goals. They are able to make choices and are characterized by behaviour. They will have to restrict this behaviour to an adequate control (reading

and/or writing) of system items in accordance with a behaviour described in the requirement document.

To be satisfied, a goal may need the cooperation of several agents; therefore the system's behaviour will correspond to the parallelization of the system's agents' behaviours. Agent behaviour is composed of a sequence of state transitions for the items under the control of the agent. Those items are state variables, corresponding to a functional pair $(x, v)$ where $x$ is the variable and $v$ its value. The system state will correspond to the aggregation of variables' states characterizing the system, meaning that goals will have to be formulated in terms of shared phenomena between agents. A phenomena will always be controlled (written) by one agent and monitored (read) by another.

An agent may be a person, a role in an organization, a device, an existing software or a software-to-be. A distinction is made between environment's agents and agents of the software-to-be. Goals under the responsibility of the former one become expectations. They express an expected behaviour from an agent of the environment needed to fulfill the parent goal. They can't be enforced by the software-to-be. Goals under the responsibility of software-to-be agents, also called system agents, become requirements. They express an expected behaviour of the software-to-be [van Lamsweerde, 2001].

### 2.2.3 Domain Properties and Hypothesis

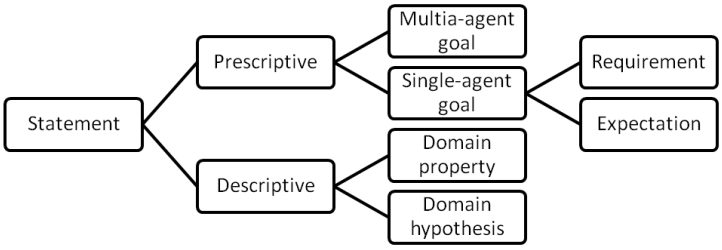


Figure 2.4: Statement typology with goals [van Lamsweerde, 2009]

Goals need agents to be fulfilled, they are expressed for a particular system in its particular environment. The intrinsic nature of that environment with its laws may naturally satisfy goals. For instance, the goal *"Getting accurate price from Supplier"* may be satisfied by the environment property *"Supplier publish accurate merchandise's prices on their website"*. Therefore, GORE also includes the notions of domain property and hypothesis.

A *domain property* is a descriptive statement that is true independent of the system, *e.g.* the speed of light is 299 792 458 m/s.

A *hypothesis* is a descriptive statement satisfied by the environment of the system, *e.g.* company is closed on Sunday.

Figure 2.4 shows a statement typology with goals.

## 2.3 KAOS: a Goal-Oriented Method

KAOS is a goal-oriented method used in the software requirements engineering process. It describes requirements using four sub-models: a goal model, a responsibility model, an object model and an operation model. KAOS stand for *Knowledge Acquisition in autOmated Specification* or *Keep All Objects Satisfied* and is supported by a tool called *Objectiver* [Respect-IT, 2009]. The KAOS language itself is described in a meta-model which also contains meta-constraints. The most important ones that will be used afterwards are presented in this section. For more informations, the interested reader could refer to one of the works this section is mainly inspired from: van Lamsweerde's work [van Lamsweerde, 2001,van Lamsweerde, 2009], Letier's thesis [Letier, 2001] or Objectiver manuals [CEDITI, 2003,Respect-IT, ,Respect-IT, 2009].

### 2.3.1 Mine Pump Example

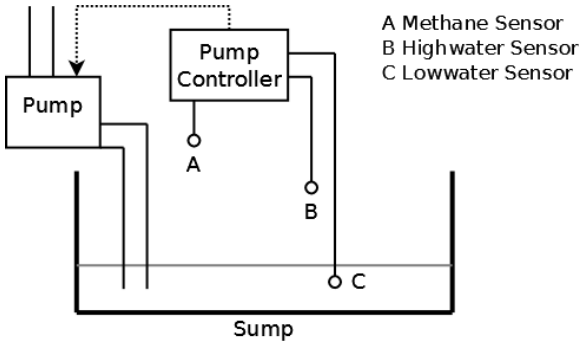


Figure 2.5: Mine Pump and Pump Controller system [Letier, 2001]

To illustrate the different points in the remainder of this work, we will use a mine pump example presented in figure 2.5 and inspired from [Letier, 2001]. In this example, we have a mine that has to be kept safe from flooding and avoid an explosion. For this we have a mine pump that starts pumping if the water level is too high and if there is no methane detected. Here is the informal problem statement:

> Water percolating into a mine is collected in a sump to be pumped out of the mine. Two water level sensors detect when water is

above a high and below a low level, respectively. A pump controller switches the pump on when the water reaches the high water level and off when it goes below the low water level. The mine has another sensor to monitor the methane level. An alarm must be raised if any of these levels becoming critical so that the mine can be evacuated. To avoid the risk of explosion, the pump must be operated only when the methane level is below a critical level.

### 2.3.2 Goal Model

The main concept in this model is the goal, represented by a blue parallelogram in diagrams, which corresponds to an objective the considered system should satisfy through the cooperation of its agents. It is formulated in a prescriptive statement at a certain level of abstraction.

The goal model will serve as a basis for other models and is usually the first one that is elaborated in a KAOS requirements elicitation process. An initial set of goals in the model can be discovered by techniques like analysing the current objectives and problems in the system-as-is, searching for goal-related keywords in elicitation material, instantiate the different goal categories (see section 2.2.1), *etc.*

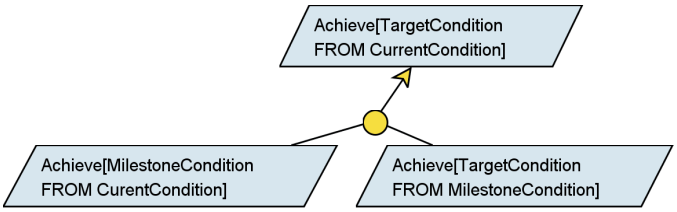


Figure 2.6: Abstract Milestone-driven refinement pattern

Once an initial set of goals are discovered, they will be abstracted and refined through new requirements elicitations. To guide this process, the analyst can use a catalogue of refinement patterns. One of the most common ones is the milestone-driven refinement, used when a target condition can be reached from a current condition with an intermediate condition, the milestone. Figure 2.6 shows the abstract definition of this pattern where the *TargetCondition*, *CurrentCondition* and *MilestoneCondition* will have to be instantiated.

#### AND/OR Graph

Goals are arranged in a AND/OR graph where goal refinement nodes, represented by a yellow circle, will be used to connect a goal, saying *why* sub-goals are needed, to a set of sub-goals saying *how* the parent goal

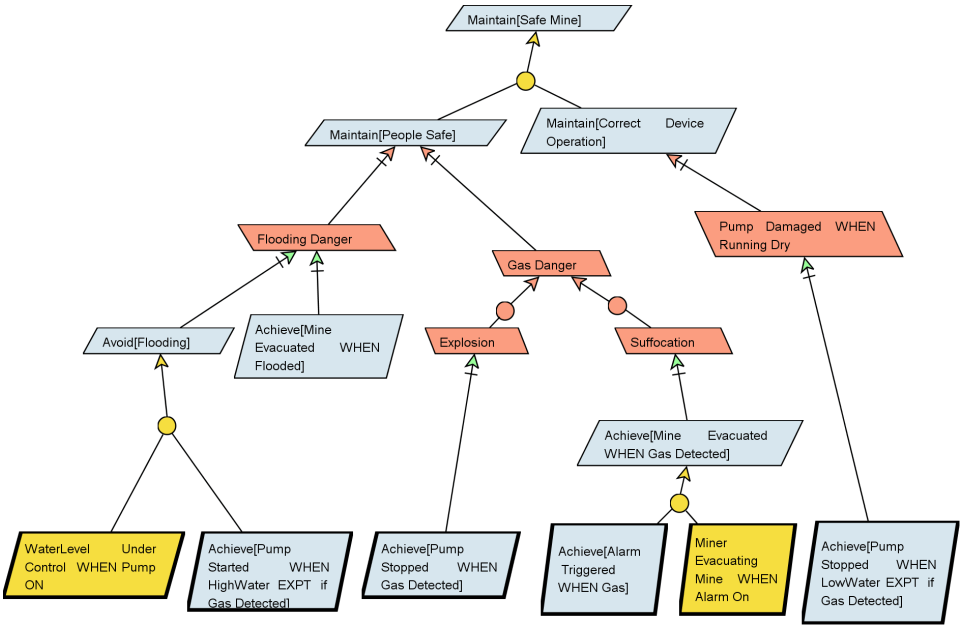


Figure 2.7: Mine pump example's goal refinement

will be fulfilled. Figure 2.7 shows an example of refinement where a goal `Maintain[SafeMine]` corresponding to a general requirement "*keeping a mine safe*", will be achieved by "*keeping people in the mine safe*" and "*keeping the devices in the mine correct and operational*".

In a goal diagram, non-leaf goal nodes correspond to OR-nodes whereas refinement nodes, represented by yellow circles, correspond to AND-nodes. In the case of a OR-node, it is said satisfied if one of its descendants is satisfied. In the case of a AND-node, it is said as satisfied if all its descendants are satisfied. For instance in figure 2.8, the parent goal `Achieve[Keep people informed]` is satisfied if "*an e-mail address is registered when the user subscribed to the service*" and "*an e-mail is send when a new event is organised*", or if "*a cellphone number is registered when the user subscribed to the service*" and "*an sms is sent when a new event is organised*".

### AND-Refinement

To check the goal model, three criteria are defined for the AND-refinement: completeness, consistency and minimality. They can be used as a tool for further elicitation and should be verified for mission critical goals by using formal techniques such as theorem proving, the use of catalogue of formal refinement patterns or SAT solver technologies.

**Criterion** (Completeness)**.** *The satisfaction of all sub-goals* $G_1, ..., G_n$ *should be sufficient for the satisfaction of the parent goal* $G$ *in view of all known do-*

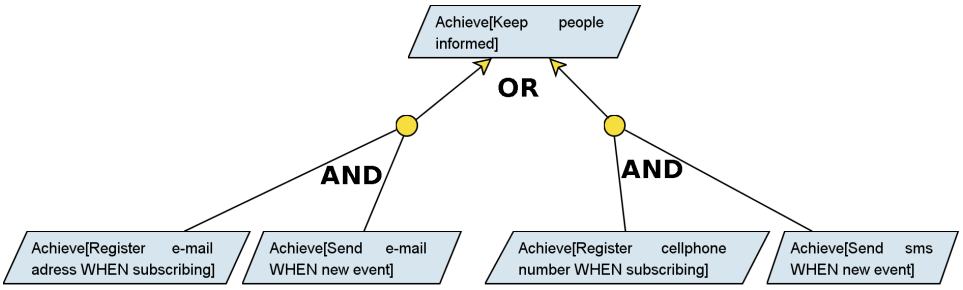


Figure 2.8: A KAOS AND/OR graph example

*main properties and hypothesis. The lattes will be represented by the set Dom.*

$$\{G_1, ..., G_n, Dom\} \models G$$

Where $A \models B$ means that $B$ is satisfied in any circumstance where all expressions in $A$ are satisfied.

**Criterion** (Consistency)**.** *The sub-goals $G_1, ..., G_n$, domain properties and hypothesis in Dom may not contradict each other.*

$$\{G_1, ..., G_n, Dom\} \not\models false$$

**Criterion** (Minimality)**.** *If one of the sub-goals $G_i$ in the refinement $G_1, ..., G_n$ is missing, the satisfaction of the parent goals $G$ is no longer always guaranteed .*

$$\forall i : 1 \leqslant i \leqslant n, \{G_1, ..., G_{i-1}, G_{i+1}, ..., G_n, Dom\} \not\models G$$

### OR-Refinement

OR-refinement is used to represent alternative options in a goal model. In this case, a parent goal can be satisfied by satisfying all sub-goals from any of the alternative refinements. Those alternatives will usually lead up to different versions of the modelled system.

The goal model with its OR-refinements will serve as a basis to evaluate the different possibilities. By discussing with the stakeholders the advantages and disadvantages, evaluating the different softgoals and their satisfaction rate if one alternative or another is taken. After negotiation with the decision takers, one alternative will be selected for the specification of the system.

### Conflicts

When elaborating the goal model, there may be divergences between the different goals. Those divergences capture potential conflicts, represented by

a red flash in KAOS, where some statement becomes logically inconsistent if a boundary condition becomes true. Roughly, goals $\{G_1, ..., G_n\}$ are divergent, given a set of domain properties and hypothesis $Dom$, if there exists a boundary condition $B$ such as:

$$\{G_1, ..., G_n, B, Dom\} \models false$$
$$\text{whereas } \{G_1, ..., G_n, Dom\} \models true$$

A more complete definition of divergence can be found in [van Lamsweerde, 2009].

Those conflicts must be resolved, but not necessary in a early phase of requirements elaboration. Indeed, they may be a source of useful information for further elicitations.

#### Agents

KAOS makes the distinction between software-to-be agents and environment agents. Both are represented as flat hexagons, with a little picture of a man into it to denote environment agents. Leaf-goals under the responsibility of those agents become expectations represented by a yellow parallelogram with a bold border. Goals under the responsibility of software-to-be agents, also called system agents, become requirements represented by a blue parallelogram with a bold border too. See subsection 2.3.5 to have more details about agents and their responsibilities.

The visual separation of environment agents and software-to-be agents, and between expectations and requirements allows to visually distinguish parts that will be implemented in the software-to-be and parts that will have to be ensured by the environment.

#### Other Concepts

The other main concepts that can be found in a KAOS goal diagram are the soft goal represented by a blue parallelogram with dotted borders, the domain property represented by a purple pentagon, the obstacles represented by a red parallelogram, the obstruction link between an obstacle and a goal and the resolution link between a goal and an obstacle. Figure 2.9 shows the graphical representation of KAOS main concepts, with the relations between elements of the goal model and elements coming from other models that will be described in the following sub-sections. More information about all those elements can be found in [Letier, 2001, van Lamsweerde, 2009].

### 2.3.3 Object Model

The object model is constructed in parallel with the goal model as soon as the latter becomes precise enough. The idea here is to identify and give a

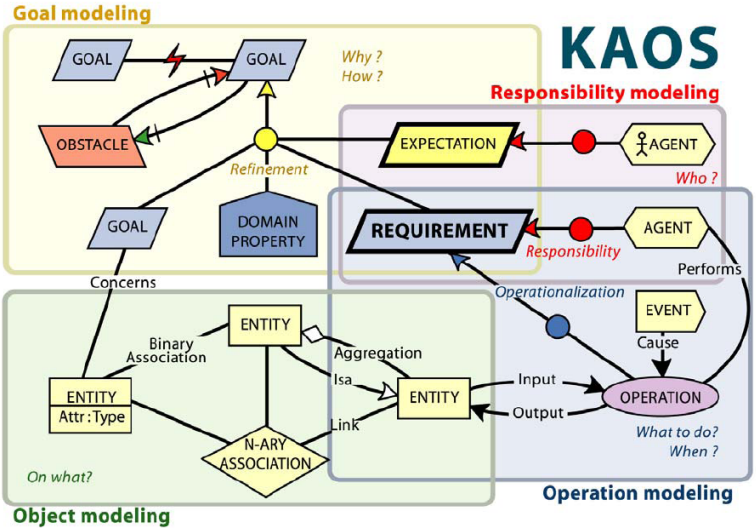


Figure 2.9: KAOS main concepts [Respect-IT, 2009]

precise definition for every object, relationship or attribute coming from the goal elicitation.

Formally speaking, an object in KAOS may be an entity that is a passive object, an association that is a subordinate object, an agent that is an active and autonomous object capable to execute operations or an event that is an instantaneous object that exists in one state only of the system. The IsA relation may be used to express inheritance between objects. The object modelling rectangle of figure 2.9 shows the graphical representation of those different elements.

Every object has a name and a definition. A set containing all the object's instances is implicitly declared for each object. If the object is an association, it defines a mathematical relationship between $n$ objects, each one will have a role that defines its function in the relation and a cardinality that defines the minimum and maximum number of instances of the association in which a given object instance can be involved simultaneously. As in UML, an association may be a simple association, an aggregation or a composition, and may be directed or not. It is important to note that the cardinalities of an association may only describe domain properties and not objectives of the system, *e.g.* "*not more than hundred books per category*" is an objective the system should meet, it will not be represented as a maximal cardinality in the object model.

The remaining entities represented in the object model are the attributes. They have a name, a definition and a range of values. An attribute is defined for an object and can be seen as a function, total or not, from the set of object instances to the set of possible values defined for the attribute.

In KAOS, a meta-constraint between the goal model and the object

model states that [Letier, 2001]:

**Meta-constraint 2.3.1** (Consistency rule between object and goal model)**.** *Every vocabulary element used in the definition of goals must be declared in the object model.*

### 2.3.4 Formal Layer Using Linear Temporal Logic

KAOS uses linear temporal logic to express formal definitions and annotations in the different models. In those formal definitions, variables correspond to arbitrary object instances, *e.g.* in $\forall tr_1 : Train$ variable $tr_1$ represents arbitrary instances of the Train entity, and functions are used to represent attributes and binary associations, *e.g.* $\forall tr_1 : Train, speed(tr_1) < MAX\_SPEED$ means that for all instances of Train, the actual speed must be lower than a $MAX\_SPEED$ constant. As recall, objects are described in the object model which regroups all the notions manipulated in the goal model.

Each variable $x$ has a value $v$ which corresponds to a tuple of values for its attributes and the relations in which the corresponding object participates. The state of a variable is defined as a functional pair $(x, v)$. In the same way, the system state is defined as a tuple $(X, V)$, where $X$ is a tuple of variables $x$ and $V$ is the tuple of corresponding values $v$ for the variables in $X$.

Linear temporal logic allows to refer to future and past states of the system by introducing the notion of history. A history $H$ is a function $H : \mathbb{N} \rightarrow State(X)$ assigning to every time point $i$ in $\mathbb{N}$ the system state at that point. $State(X)$ is the set of all possible values for the variables in $X$.

**Definition 2.3.1** (Temporal assertion satisfaction)**.** *If a temporal assertion $P$ is satisfied by a history $H$ at time position $i$, we say that :*

$$(H, i) \models P$$

*If $i$ is the initial position 0, then the assertion $P$ is said to be satisfied by the entire history $H$ :*

$$(H, 0) \models P$$

Correctness of the definitions and annotations will have to be satisfied by the history $H$ of the system at a certain time position $i$. Those definitions and annotations may be expressed as *state assertions*, that is a predicate which is true in the current state, but will more probably be a temporal assertion, recursively build from state assertions, temporal operators, logical connectives and quantifiers.

### State Assertions

State assertions are build from atomic predicates connected through classical logic connectors (and $\wedge$, or $\vee$, not $\neg$, implies $\rightarrow$, and equivalent to $\leftrightarrow$) and quantifiers (for all $\forall$ and there exists $\exists$). The atomic predicates are composed of terms connected by relational operators, *e.g.* greater than or equal $\geqslant$, or associations defined in the object model, *e.g.* $Borrows(b1, c1)$ where $b1$ is an instance of Book and $c1$ an instance of Client. Terms are built from constants, variables and function symbols applied to terms. Those functions can be mathematical functions, like the arithmetic operators, or attributes declared in the object model.

For instance, a security requirements stating that *the water level in a mine must always be lower than a maximal level* could be translated in :

$$\forall m_1 : Mine$$
$$waterLevel(m_1) < MAX\_WATER\_LEVEL$$

Assuming that $waterLevel$ is an attribute of the entity $Mine$ and $MAX\_WATER\_LEVEL$ is a constant value.

### Temporal Assertions

Temporal assertions are build recursively from state assertions, temporal operators, logical connectives and quantifiers. Contrary to state assertions, they do not only refer to the current state but also to previous and future states. The tables 2.1 and 2.2 summarize the time operators used with KAOS and they associated semantics.

Table 2.1: Future time operators

| Notation | Informal Explanation | Semantic |
|---|---|---|
| $\Diamond P$ | Sooner or later $P$ | $(H,i) \models \Diamond P$ **iff** $\exists j, j \geqslant i : (H,j) \models P$ |
| $\Box P$ | Always $P$ | $(H,i) \models \Box P$ **iff** $\forall j, j \geqslant i : (H,j) \models P$ |
| $P\ \boldsymbol{U}\ Q$ | Always $P$ until $Q$ | $(H,i) \models P\ \boldsymbol{U}\ Q$ **iff** $(\exists j, j \geqslant i : (H,j) \models Q) \wedge (\forall k, i \leqslant k < j : (H,k) \models P)$ |
| $P\ \boldsymbol{W}\ Q$ | Always $P$ unless $Q$ | $(H,i) \models P\ \boldsymbol{W}\ Q$ **iff** $((H,i) \models P\ \boldsymbol{U}\ Q) \vee ((H,i) \models \Box P)$ |
| $\circ P$ | Next $P$ | $(H,i) \models \circ P$ **iff** $(H, i+1) \models P$ |
| $P \Rightarrow Q$ | $P$ entails $Q$ | Equivalent to $\Box(P \rightarrow Q)$ |
| $P \Leftrightarrow Q$ | $P$ is congruent to $Q$ | Equivalent to $\Box(P \leftrightarrow Q)$ |

Table 2.2: Past time operators

| Notation | Informal Explanation | Semantic |
|---|---|---|
| $\blacklozenge P$ | Once $P$ | $(H,i) \models \blacklozenge P$ **iff** $\exists j, j \leqslant i : (H,j) \models P$ |
| $\blacksquare P$ | $P$ has always been | $(H,i) \models \blacksquare P$ **iff** $\forall j, j \leqslant i : (H,j) \models P$ |
| $P \ \boldsymbol{S} \ Q$ | Always $P$ in the past since $Q$ | $(H,i) \models P \ \boldsymbol{S} \ Q$ **iff** $(\exists j, j \leqslant i : (H,j) \models Q) \wedge (\forall k, j < k \leqslant i : (H,k) \models P)$ |
| $P \ \boldsymbol{B} \ Q$ | Always $P$ in the past back to $Q$ | $(H,i) \models P \ \boldsymbol{B} \ Q$ **iff** $((H,i) \models P \ \boldsymbol{s} \ Q) \vee ((H,i) \models \blacksquare P)$ |
| $\bullet P$ | Previously $P$ | $(H,i) \models \bullet P$ **iff** $(H, i-1) \models P$ with $i > 0$ |
| $@P$ | To $P$ | Equivalent to $(\bullet \neg P) \wedge P$ |

For example, a requirement saying that "*for a mine, an alarm has to be triggered as soon as methane is detected in the mine*" could be translated in:

$$\forall m_1 : Mine$$
$$methane(m_1) = true \Rightarrow \circ\, bell(m_1) = ON$$

Assuming that *methane* and *bell* are attributes of the $Mine$ entity and $ON$ is a constant value.

### Bounded Time Operators

Requirements like the one in the example here are rare, a more realistic version of it may be: "*for a mine, an alarm has to be triggered in the three seconds following methane detection*". Although such bounded time operators are not present in classical temporal assertions. To express such requirements, KAOS uses three kinds of bounded time operators.

**Relative time bound** refers to a time distance from the current state. The requirement given above is an example of such time bound. To define such relative time bounds, a temporal distance function between states must be introduced:

$$dist : \mathbb{N} \times \mathbb{N} \rightarrow D \text{ where } D = \{d | \exists n : d = n \times u\}$$
$$dist(i,j) = |j - i| \times u$$

Where $u$ corresponds to the chosen time unit, *e.g.* second, microsecond, days, week, *etc.* Note here that if multiple time unit are used, they are implicitly converted into the smallest one. Time operator $\Diamond_{\leqslant d} P$ will correspond to *Sooner or later within deadline d, P*. The semantics becomes:

$$(H,i) \models \Diamond_{\leqslant d} P \textbf{ iff } \exists j, j \geqslant i \wedge dist(i,j) \leqslant d : (H,j) \models P$$

where $d \in D$. Our requirement " *for a mine, an alarm has to be triggered in the three seconds following methane detection* " will be translated into:

$$\forall m_1 : Mine$$
$$methane(m_1) = true \Rightarrow \Diamond_{\leqslant 3\,sec.}\, bell(m_1) = ON$$

With a unit time $u$ corresponding to one second.

**Absolute time bound** is used for requirements that refer to an absolute time system, *e.g.* "*Book copies shall be returned by the end of the year for inventory*". To do this, every time point of the system has to be associated to the actual time in $Time$. It is done by a *clock* function:

$$clock : \mathbb{N} \rightarrow Time$$
$$clock(i) = clock(0) + dist(0,i)$$

Time operator $\Diamond_{\leqslant ct} P$ will correspond to *Sooner or later before clock time ct,* $P$. The semantics becomes:

$$(H,i) \models \Diamond_{\leqslant ct} P \textbf{ iff } \exists j, j \geqslant i \wedge clock(j) \leqslant ct : (H,j) \models P$$

Where $ct \in Time$.

**Variable dependant time bound** allows to refer to state variables of the object model, attributes or associations, whose values may change over time. Those kinds of variable time bounds are generally the most used in requirements, *e.g.* "*For every cinema, reservations are closed three hours before the beginning of the projection*" refers to projections $p$ in a cinema and to the beginning of the projection $p.Time$. Variables may express relative time bounds, like a delay, or absolute time bounds, like a fixed date. The two functions $dist(i,j)$ and $clock(i)$ will be used to define such time bounds. For a state variable $sv$ expressing a delay the semantics of the operator $\Diamond_{\leqslant d(sv)} P$ is:

$$(H,i) \models \Diamond_{\leqslant d(sv)} P \textbf{ iff } \exists j, j \geqslant i : (H,j) \models P$$
$$\text{And } \forall k, i \leqslant k \leqslant j : clock(k) - clock(i) \leqslant VAL_{H,k}(d(sv))$$

Where $VAL_{H,k}(d(sv))$ corresponds to the value of the variable dependent deadline $d(sv)$ at time position $k$ along history $H$

For a state variable $sv$ expressing a fixed real time point the semantics of the operator $\Diamond_{\leqslant ct(sv)} P$ is:

$$(H,i) \models \Diamond_{\leqslant ct(sv)} P \textbf{ iff } \exists j, j \geqslant i : (H,j) \models P$$
$$\text{And } \forall k, i \leqslant k \leqslant j : clock(k) \leqslant VAL_{H,k}(ct(sv))$$

Where $VAL_{H,k}(ct(sv))$ corresponds to the value of the variable dependent clock time $ct(sv)$ at time position $k$ along history $H$

### 2.3.5 Responsibility Model

The responsibility model presents the different agents of the system and their responsibility in terms of desired behaviours. Declaring a responsibility assignment of a goal to an agent intuitively means that the agent is the only one required to restrict its behaviour so as to ensure the goal. Responsibility assignments are graphically represented by a link with a red circle. They provide a criterion for stopping the goal refinement process. As stated before, a goal assigned to a software agent becomes a requirement, while a goal assigned to an environment agent becomes an expectation, also sometimes called assumption.

**Criterion** (Stopping goal refinement process)**.** *A goal assigned as the responsibility of a single agent must not be refined further.*

#### Control and Monitor Links

The intuitive meaning of a responsibility assignment says that the agent behaviour is able to fulfil the assigned goal. It means that the agent is able to read and modify elements of the object model according to the definition of the goal. On the other side an agent may be responsible for a goal if and only if it has the capabilities to read and write elements used in the goal's definition. Those capabilities are captured in the model through *Monitor* and *Control* links. The former links an agent to an element of the object model if the agent can read this element. The second links an agent to an element of the object model if the agent can modify this element. To avoid interference problems between concurrent executions of agents, the following meta-constraint is defined.

**Meta-constraint 2.3.2** (Single control)**.** *An element of the object model may be controlled by at most one agent.*

Agent behaviours are made more explicit in the *operation model* where the operations needed to fulfil a goal are declared and linked to the goal through *operationalization links*. A meta-constraint states that the agent responsible for the goal will be the one performing those operations that *operationalize* that goal.

#### Agent Diagram

Agent's capabilities and responsibilities are presented in an agent diagram. For instance, figure 2.10 presents the agent diagram of a pump controller, used in a mine to avoid flooding. The *PumpController* is monitoring the $Mine.waterLevel$ and $Mine.methane$ attributes and controlling the $Mine.pump$ attribute. According to the water level and the presence or not of methane, the *PumpController* will launch or stop the pump. The pump controller

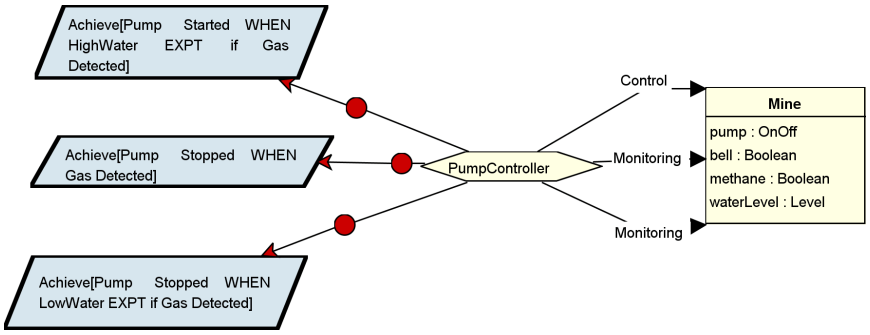


Figure 2.10: Agent diagram: pump controller

example will be more systematically introduced in the beginning of chapter 4.

### Context Diagram

Another kind of diagram can be used to present both agent capabilities and how information will flow in the system. The context diagram is sometimes used in an early phase of the elicitation process, *e.g.* a management system where information flows from one agent to another. It also usually facilitates the responsibility assignment. Figure 2.11 presents the context diagram for the mine pump example where according to some sensor's data a pump is switched on or off and an alarm may be triggered to notify miners of the presence of methane in the mine.

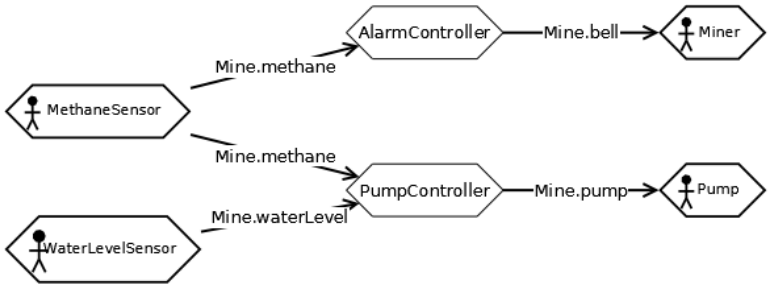


Figure 2.11: Context diagram: mine pump

## 2.3.6 Operation Model

As explained in the previous sections, an agent will have a declared behavior corresponding to a sequence of state transitions for the object attributes and associations that the agent controls. Those transitions correspond to executions of operations performed by the agent.

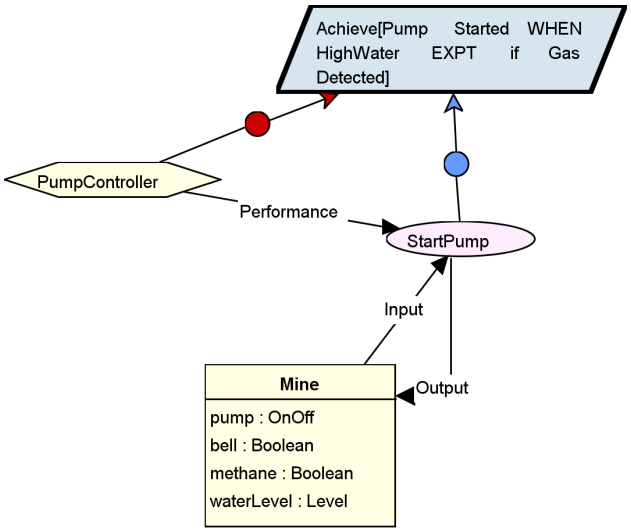


Figure 2.12: Starting the pump operation model

The operation model presents the operations which have a name, a definition, a domain pre-condition and a domain post-condition with their inputs and outputs and the agent that performs it. The domain pre-condition of an operation characterizes the input states when the operation is applied and the domain-post-condition of an operation characterizes the class of output states when the operation has been applied. Both pre-conditions and post-conditions do not care about required conditions for goal satisfaction, *e.g.* for an operation *StartPump*, the domain pre-condition will be $m : Mine, m.pump = off$ and the domain post-condition will be $m : Mine, m.pump = on$, the condition here does not care if methane is detected before starting the pump or not.

An operation is linked to a leaf goal by an operationalization link and to an agent by a performance link with the meta-constraint:

**Meta-constraint 2.3.3** (Responsibility)**.** *The agent performing the operation must be responsible of the operationalized goal.*

On a diagram, as in figure 2.12, it will correspond to the two links, on operationalization link with a blue circle and a responsibility link with a red circle, linked to the same leaf-goal. Note here that an agent performing an operation has to have the ability to monitor the inputs of the operation and control its outputs.

**Meta-constraint 2.3.4** (Input/Output)**.** *The agent performing the operation must have capability to monitor the inputs and control the output of the operation.*

For convenience reasons, those links are not always explicitly represented in the model but become implicit when an agent is linked to an operation through a performance link.

### Required Conditions for Operationalization Link

To ensure goal satisfaction by executing the operations, an operationalization link, linking an operation to a leaf goal, has three associated conditions:

- *Required pre-condition* which is a necessary condition expressed over the input states for the application of the operation to satisfy the linked goal. If this condition and the domain pre-condition are true, the operation *may be* executed. For example the pre-condition associated to the *StartPump* operationalization link in figure 2.12 will be $m : Mine, m.methane = false$.

- *Required trigger-condition* which is a sufficient condition expressed over the input states for the application of the operation to satisfy the linked goal. If this condition and the domain pre-condition are true, the operation *has to be* executed. For example the trigger-condition associated to the *StartPump* operationalization link in figure 2.12 will be $m : Mine, m.waterLevel = high \wedge m.methane = false$.

- *Required post-conditions* which are an additional effect of the operation must have to fulfil the operationalized goal.

A meta-constraint says that:

**Meta-constraint 2.3.5** (Required conditions)**.** *For an operationalization link, required trigger-condition implies required pre-condition.*

If this constraint was not fulfilled, it could lead to required trigger-condition and domain pre-conditions being both true while the required pre-condition is false. The meta-constraint can be respected by simply replacing the required trigger-condition by the conjunction of required trigger-condition and required pre-condition.

An operation can also operationalize more than one leaf goal. In this case, if the domain pre-condition holds, the operation may be executed if *all* of its required pre-conditions are true and as-soon-as *one* of its trigger-conditions is true. The global required trigger-condition becomes the disjunction of all the required trigger-conditions and the global required pre-condition becomes the conjunction of all required pre-conditions. According to the required conditions meta-constraint, the global required trigger-condition has to imply the global required pre-condition.

The interpretation of the operation model is as follows:

*For a goal G under the responsibility of an agent A,*

*for every operation O operationalizing G, agent A must guarantee that operation O is applied when O's domain pre-condition holds*

- *as soon as one of O's required trigger-condition holds and only if all O's required pre-conditions hold,*
- *so as to establish O's domain post-condition together with all O's required post-conditions.*

**Formal Interpretation**

Operations, like other elements of the KAOS language, may be formally defined using linear temporal logic. This may be interesting to do model checking, particularly for critical components.

An Operation *op* corresponds to relations between input and output variables according to the domain pre-condition and the domain post-condition:

$$[|op|] =_{def} DomPre(op) \wedge \circ DomPost(op)$$

Where $[|c|]$ is a notation corresponding to the linear temporal formula giving the semantics of $c$, and $DomPre(op)$ and $DomPost(op)$ are $op$'s domain pre-condition and post-condition.

In the same way, the definition of $op$'s required pre-conditions $ReqPre$, required trigger-conditions $ReqTrig$ and required post-conditions $ReqPost$ are:

$$R \in ReqPre(op) : [|R|] =_{def} (\forall *)[|op|] \Rightarrow R$$
$$R \in ReqTrig(op) : [|R|] =_{def} (\forall *)DomPre(op) \wedge R \Rightarrow [|op|]$$
$$R \in ReqPost(op) : [|R|] =_{def} (\forall *)[|op|] \Rightarrow \circ R$$

Where $(\forall *)$ means that all free variables in its scope are universally quantified. Remember here that in the context of linear temporal logic formula, $P \Rightarrow Q$ is equivalent to $\Box(P \rightarrow Q)$.

For example, for the *StartPump* operation:

- $m : Mine,\ DomPre(StartPump) \equiv m.pump = off$
- $m : Mine,\ DomPost(StartPump) \equiv m.pump = on$
- $m : Mine,\ ReqPre(StartPump) \equiv m.methane = false$
- $m : Mine,\ ReqTrig(StartPump) \equiv m.waterLevel = high \wedge m.methane = false$

The semantics of the operation is then:

- $m : Mine,\ [|StartPump|] =_{def} m.pump = off \wedge \circ m.pump = on$

For the required condition, we have that:

- $m : Mine,\ [|ReqPre|] =_{def} (m.pump = off \wedge \circ m.pump = on) \Rightarrow m.methane = false$
- $m : Mine,\ [|ReqTrig|] =_{def} (m.pump = off \wedge m.waterLevel = high \wedge m.methane = false) \Rightarrow (m.pump = off \wedge \circ m.pump = on)$

As for goal refinement, completeness, consistency and minimality criteria are defined to check whether a set of operations correctly operationalize a requirement or an expectation.

**Criterion** (Correct goal operationalization). *Let $R_1, ..., R_N$ be the required conditions defined on the operations operationalizing a goal $G$. This operationalization is correct if and only if:*

$[|R_1|], ..., [|R_n|] \models G$ *(Completeness)*

$[|R_1|], ..., [|R_n|] \not\models false$ *(Consistency)*

$G \models [|R_1|], ..., [|R_n|]$ *(Minimality)*

This criterion may be used to verify the operationalization, *e.g.* with a SAT solver to find counterexample for formula $[|R_1|] \wedge ... \wedge [|R_n|] \wedge Dom \wedge \neg G$.

#### Difference Between Goals and Operations in KAOS

In KAOS, there is a difference between goals and operations. Both express constraints over system state transitions, but a goal expresses constraints on sequences of transitions while an operation expresses constraints on a single transition.

It is important to understand that an operation is not "something" that leads the system from one state $A$ to another state $B$, but a restriction on all the possible system state transitions from $A$ to $B$ to those permitted by the operation.

### 2.3.7 KAOS's Supporting Tool

*Objectiver* [Respect-IT, 2009] is a tool that supports the KAOS method. The four sub-models are present in it with other tools like a query editor to interrogate a model with a SQL-like syntax, a use-case model generator to generate UML use case diagrams [OMG, 2009a,OMG, 2009b] from operation models and an EMF connector to permit external applications to connect to *Objectiver* and get the currently edited KAOS model in a XMI format[1].

[1] This last feature has been used in our prototype presented in section 4.8 where EMF and XMI are also explained

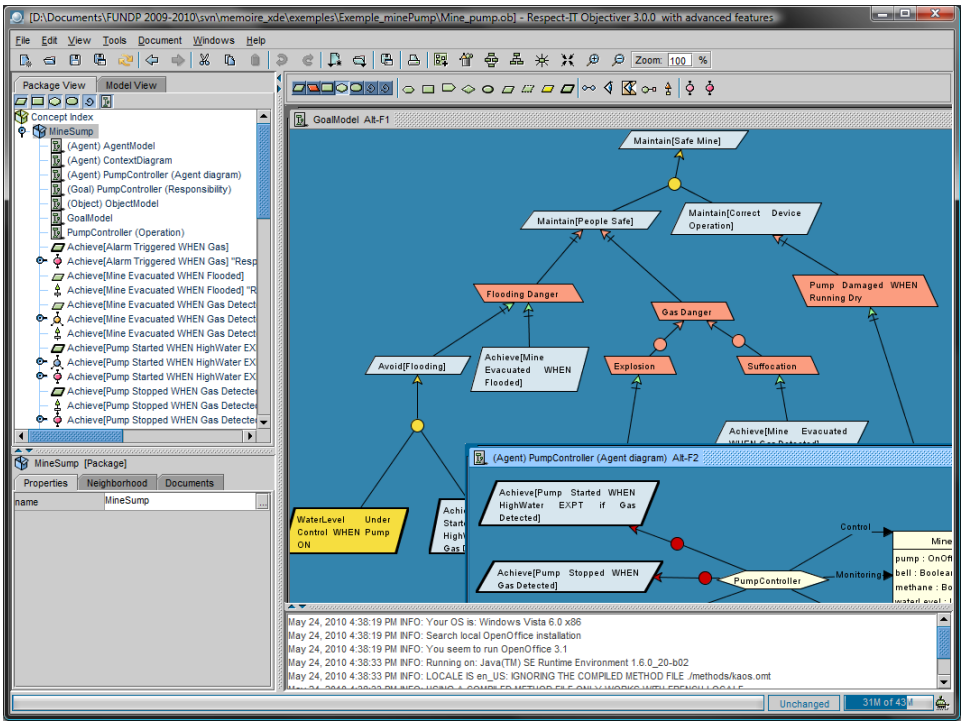


Figure 2.13: *Objectiver* print screen of the mine pump example [Respect-IT, 2009]

Figure 2.13 shows a print screen of the mine pump example encoded with *Objectiver*. The concept index tree one the top left shows all the elements present in the KAOS model. The diagram editor on the right part shows the currently edited *Goal model* and *PumpController agent* diagrams. The property editor on the bottom left part of the print screen is used to edit properties of the concept currently selected in the concept index tree.

# Chapter 3

# Formal Modelling for Specifications

This chapter introduces the notion of formal modelling with, in the first section, a general overview of formal methods. The second section focus on a particular formal method, Event-B, used to model systems as discrete transition systems.

## 3.1 Overview of Formal Methods

Formalization has been steadily growing in computer sciences for years, first used for program specifications, it is now used for specifications in the large as in Event-B which can, as we will see, be used to specify a complete system and its context. Before going into details, we give a short overview of the current formal method scene inspired from [Abrial, 2010, Ball, 2008, Clarke and Wing, 1996, van Lamsweerde, 2000, Monin and Hinchey, 2003].

Formal methods aim at producing formal specifications which are defined as "*the expression, in some formal language and at some level of abstraction, of a collection of properties some system should satisfy*" [van Lamsweerde, 2000]. The properties that should be satisfied will classically be discovered during the requirements phase. The specifications are expressed in a formal language composed of syntax with a semantics relying on a mathematical substratum and a proof theory defining rules for inferring useful information from the specification. For example, in the elaboration of a compiler, the used language for describing a grammar may be BNF, relying on the theory of formal languages and automata. The benefit of a rigorous semantics is the absence of ambiguities and thus a better communication between the stakeholders implied in a system and its development. Note that it may be useful to use multilingual specifications, with good consistency management, to address the different classes of consumers and their backgrounds.

Formal specifications are precise and may be formally verified to ensure

consistency and correctness. Pay attention here that *precise* does not mean *formal*. A specification can be precise and yet not formally verifiable [van Lamsweerde, 2000]. In the classical development process, if a mistake is discovered in a "late-phase", *e.g.* bugs in the final product are typically such mistakes; this mistake will generally be more difficult to repair. This is why reliable specifications are so important [Monin and Hinchey, 2003]. Especially if a failure in the system may have important consequences, going from recovery issues to money or even human lives loss. By having consistency and correctness being proved for a given specification, a lot of mistakes may be avoided; for instance mistakes due to inattention, mistakes coming from bad reasoning, *etc.* Moreover, being precise, the specification is non-ambiguous and mistakes due to bad communication between different stakeholders can be avoided, assuming that the different stakeholders understand the semantics of the formal language used to write down the specification of the system.

Verifications in formal methods can be classified into two general families [Clarke and Wing, 1996]. The first one, called model checking, will verify that a desired property holds in a finite model. This verification in made by an exhaustive state space search that is guaranteed to finish since the model is finite. The second one, called theorem proving, will find a proof for a property from the description of the system. This may be done using axioms and inference rules of the mathematical logic in which the system and desired properties are expressed. Each step in the proof appeals to those axioms and rules, and possibly derives definitions and intermediate lemmas.

The choice of a formal method will depend on several factors, like the system scope, the kinds of property, the level of abstraction, *etc.* A classification of formal specifications, according to the paradigm they rely on, is proposed by van Lamsweerde in [van Lamsweerde, 2000] :

- A *history-based specification* presents the maximal set of admissible histories of a system, its behavior, over time. It uses time operators to express temporal logic assertions about system objects. For example the formal layer used in KAOS.

- *State-based specifications* describe the admissible states for a system at some arbitrary snapshot. The properties are expressed in invariants, constraining the system objects at any snapshot, and pre- and post-assertions constraining the application of system operations at any snapshot. Pre-assertions capture weakest necessary conditions on input states for an operation to be applied, while post-assertions capture strongest effect conditions on output states if an operation is applied. This category contains languages such as Z [ISO/IEC, 2002], VDM [Jones, 1990] or B [Abrial, 1996,Schneider, 2001], the ancestor of Event-B.

- *Transition-based specifications* describe required transitions from state to state. The properties are specified by a set of transition functions in a state machine. A transition function for a system object gives, for each input state, triggering event and eventually pre-condition, the corresponding output state. Statecharts [OMG, 2009b, p. 525] are included in this category of formal specifications.

- *Algebraic specifications* specify a system as a structured collection of mathematical functions. This collection usually contains constructor functions, used to create simple elements, and additional functions having a definition based on the constructor functions. For example, an algebraic specification of the Booleans will have $true$ and $false$ as constructor functions and all other classical operators like $\wedge$, $\vee$, *etc.* as additional functions. Languages based on the Common Algebraic Specification Language [Bidoit and Mosses, 2004], a specification language constructed with the aim to subsume many existing specification languages and based on first-order logic with induction, enter in this category.

- *Operational specifications* characterize a system as a structured collection of processes that can be executed by some more or less abstract machine. Petri net and process algebras [Hoare, 1985] belong to this category.

The number of developments using formal methods, and success stories going with them, is growing each year, and contrary to the commonly accepted idea, the cost spend to obtain such higher quality products decreased [van Lamsweerde, 2000]. Despite that fact, the main lack in formal methods is the absence of constructive methods for building correct specifications for complex systems in a safe, systematic and incremental way. Actual techniques generally pay no attention to the upstream of the software lifecycle and the products, like the requirements document, from which the formal specification is coming.

## 3.2 Event-B: a Formal Specification Language for System Design

The goal of formal modelling techniques is to specify an unambiguous system in such a way that it could be formally verified to guarantee consistency and correctness. Such a specification will, in case of Event-B, lead to a coded system correct by construction, where the final code will be the result of a process starting from a very general model that will be refined into more detailed model, refined in its turn, *etc.*

Event-B is one of those techniques used to model discrete transition systems. This section is largely inspired by Abrial's and Baal's works [Abrial, 2010,Ball, 2008], the RODIN deliverable 3.2 [Métayer et al., 2005] and Robinson's concise Event-B summary [Robinson, 2009].

### 3.2.1 General Overview

Event-B is issued from a simplification and an extension of the B-Method, also called Classical-B [Abrial, 1996, Schneider, 2001]. Both of them are mathematical approaches for developing formal models of systems.

Event-B models are discrete models made of states, represented by state variables, constants and invariants over these variables, and transitions activated under certain conditions. Transitions, also called events, are conditioned by a guard constructed over the variables and constants, representing the necessary condition for the occurrence of the event and defined by actions that describe how the variables will be modified after the occurrence of the event. The variables and constants definitions and manipulations rely on set theory, while conditions are expressed using propositional and predicate calculus. A model will describe the active part in a *machine*, with the variables, invariants and events, but it will also describe the environment or *context* of this active part, with the static properties of the system. The model is thus a closed model able to exhibit actions and reactions between a machine and its context.

An event, which is defined as an observable transition of state variables takes no time. As a direct consequence, two events can't occur at the same time. An informal execution interpretation of an Event-B model can be the next:

- If no guards are true, the model execution stops and the system is said deadlocked.
- If one or more guards are simultaneously true, one corresponding event may occur and the state is modified accordingly to the actions defined for the event.

This introduces some kind of non-determinism and in Event-B, no assumptions are made concerning the chosen event when more than one guard is simultaneously true. If the model has at most one guard true at all time, it is said to be deterministic. An Event-B model execution does not have to finish and may run forever.

To manage the complexity, models will be constructed incrementally thanks to abstraction and refinement, starting from an abstract model gradually refined into a more precise and concrete model. This refinement process will introduce more and more variables in the model. To handle this, the abstract model may be decomposed into independent parts. The refinement

process will guaranty that the concrete model is coherent with the abstract model.

The Event-B method uses proof obligations to check consistency and correctness of the model, *e.g.* proving the coherence between an abstract model and its refinement. Those prove obligations may be generated and partially or completely proved by an automated tool [RODIN, 2010]. To do this, the properties of the specification will be used and failures will give indications about what may be wrong in the model. There are two large families of proof obligations:

- The invariants preservation property which states that under those invariants and the event guards, the invariants still hold after execution of the events actions, or in other words conditions over state variables have to be always true.
- The second family is the reachability property which states that events where the guard is not necessary true will be executed in a certain finite period.

More informations about proof obligations will be given in subsection 3.2.3.

### 3.2.2 Machines and Contexts

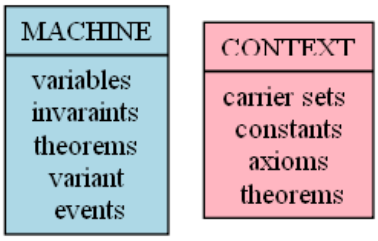


Figure 3.1: Machine and context structures

Event-B models are constructed using two kinds modelling elements, machines and contexts. Machines are used to represent the active part of the system, while context can be used to parametrize the model, *e.g.* attribute's domains or entity instances in the real world. A model containing only contexts will represent a pure mathematical structure. If the model only contains machines it means that it is un-parametrized. Classic Event-B models mix both machines and contexts, linked together like in figure 3.2.

The link between two machines is a refinement link, describing the fact that the refining machine is more concrete that the refined one. A machine can only refine at most one other machine and putting such a link will add proof obligations to the model.

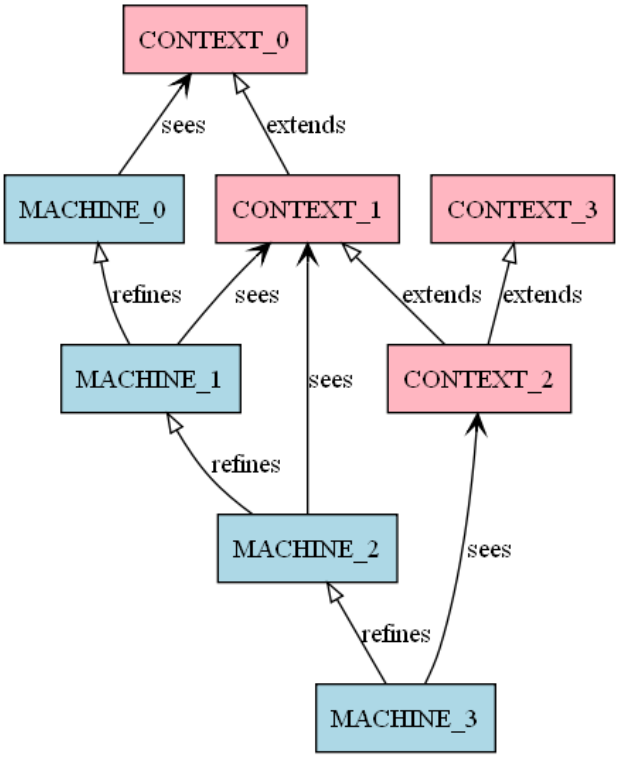


Figure 3.2: Machines and contexts links

Contexts can be linked to zero, one or more contexts through extension links, meaning that all the sets and constants of the extended contexts can be used in the extending context *e.g.* in figure 3.2 the context `CONTEXT_2` can use sets and constants defined in contexts `CONTEXT_1` and `CONTEXT_3`. This notion is transitive, *e.g.* `CONTEXT_2` can also use sets and constants from `CONTEXT_0`.

A Machine may see zero, one or more contexts, meaning that the constants and sets defined in those contexts can be used in the machine. Like the transitivity of context extension links, a machine can implicitly see all the contexts extended by an explicitly seen context, *e.g.* in figure 3.2 machine `MACHINE_3` can use sets and constants from `CONTEXT_1`, `CONTEXT_2` and `CONTEXT_3`. If the machine refines another machine, the number of its explicitly or implicitly seen contexts must be as large as the one of the refined machine, *e.g.* the link between `MACHINE_2` and `CONTEXT_1` is necessary because of the link between `MACHINE_1` and `CONTEXT_1`. Put together, the "extends" and "refines" relationships must not contain any cycle.

### Context

Figure 3.3 shows the general structure for contexts in Event-B models. Every context has a name which is unique in a model and a list of extended contexts with zero, one or more identifiers corresponding to the extended contexts names.

**CONTEXT** <name>
**EXTENDS** <context_identifier_list>
**SETS**
< set_identifier_list >
**CONSTANTS**
< constant_identifier_list >
**AXIOMS**
< label >: < *predicate* >
**THEOREMS**
< label >: < *predicate* >
**END**

Figure 3.3: Context structure

The SETS clause introduce the names list of carrier sets which defines pairwise disjoint types. Those carrier sets are not empty and can be deferred or enumerated by constants. If the carrier set is an enumerated set, the enumeration will be declared by an axiom over the set and constants accessible in the machine, *e.g.* the BOOL set provided by default contains TRUE and FALSE and represents the Boolean domain. The constant identifiers introduced by the CONSTANTS clause are unique in the context and all extended contexts, *e.g.* TRUE and FALSE are constants provided by default.

The axioms have a label and a predicate formulated over constants and sets *e.g.* the expression partition(BOOL,{TRUE},{FALSE}) means that the BOOL set is partitioned into two subsets: {TRUE} and {FALSE}. In other words, $BOOL = \{TRUE, FALSE\} \wedge TRUE \neq FALSE$. Those predicate will serve as hypotheses in all proof obligations.

Contrary to axioms, theorems are propositions that have to be proved, using the local axioms, axioms and theorems from extended contexts and theorem that have been proved before the theorem to be proved. The labels of axioms and theorems have to be unique.

Here is an example of context for the mine pump briefly presented in the previous chapter. A more complete version of this context will be described in chapter 4.

Listing 3.1: Mine pump context

**CONTEXT** MineContext
**SETS**
ONOFF, LEVEL
**CONSTANTS**
ON, OFF, LOW, MEDIUM, HIGH
**AXIOMS**
axm1 : $partition(ONOFF, \{ON\}, \{OFF\})$
axm2 : $partition(LEVEL, \{LOW\}, \{MEDIUM\}, \{HIGH\})$
**END**

### Machine

Figure 3.4 shows the general structure for machines in Event-B models. Every machine has a name which is unique in a model and may refine another machine, identified by its name. A machine can see zero, one or more contexts and use the constants and sets explicitly or implicitly defined in those contexts.

**MACHINE** <name>
**REFINES** <abstract_machine_identifier>
**SEES** <context_identifier_list>
**VARIABLES**
< variable_identifier_list >
**INVARIANTS**
< label >: < *predicate* >
**THEOREMS**
< label >: < *predicate* >
**EVENTS**
<event_list>
**END**

Figure 3.4: Machine structure

The clause VARIABLES introduces the list of variables of the machine. Their names are all distinct, but unlike contexts, some variables may be the same as some variables in the abstract machine if they have the same name. Variables may appear in invariants predicates. When a variable of the abstract machine appears in an invariant of the concrete machine, this invariant is said to be a glueing invariant, glueing the state space of concrete machine to the one of the abstract machine. As for the context, theorems of the machine will have to be proved, using the axioms and theorems of seen

contexts, invariants and theorems of the abstract machine, and invariants and theorems that have been proved before the theorem to be proved in the machine. As for context, labels of theorems and invariants have to be unique.

In Event-B machines, one can also use the notion of `VARIANTS`, but this will not be explored in the remainder of this work. We refer the interested reader to [Abrial, 2010].

Here is an example of machine for the mine pump briefly presented in the previous chapter. A more complete version of this machine will be described in chapter 4.

Listing 3.2: Mine pump machine

**MACHINE** PumpControllerMachine
**SEES** MineContext
**VARIABLES**
`pump`
`bell`
`methane`
`waterLevel`
**INVARIANTS**
`inv1` : $pump \in ONOFF$
`inv2` : $bell \in BOOL$
`inv3` : $methane \in BOOL$
`inv4` : $waterLevel \in LEVEL$
**EVENTS**
**END**

### Event

Events are introduced in a machine by the `EVENTS` keyword. Figure 3.5 gives the general form of an event. An event has a name, a status and may refine one or more events of the abstract machine if there is one. Generally, status is omitted for ordinary events.

The `any` clause introduces zero, one or more parameters for the event. It corresponds to a universally quantified new variable. Guards, which are the necessary conditions for the event to occur, follow the `where` clause. Note that this clause is sometimes replaced by `when` if there is no parameter for the event.

When an event refines another one, parameters of the abstract event introduced by the clause `any` may disappear in the concrete event. In this case, a witness has to be provided for each parameter which is initialized with a non-deterministically chosen value in the abstract event, *e.g.* a parameter taking a non-deterministic value in a set that has disappeared in the concrete event to set the value of this parameter in the concrete event. A witness must also be provided if a variable of the abstract machine that has disappeared in the concrete machine, is used in the abstract event, *i.e.* if the variable is

**Event** <*event_identifier*> $\widehat{=}$
**Status** { ordinary , convergent , anticipated }
**extends** <*event_identifier_list*>
**any**
< parameter_identifier_list >
**where**
< label >: < predicate >
**with**
< label >: < witness >
**then**
< label >: < action >
**end**

Figure 3.5: Event structure

present after the `VARIABLES` clause in the abstract machine and is used in the event of the abstract machine and is absent after the `VARIABLES` clause in the concrete machine. This witness is indicated after the `with` clause and is constituted of a label and a predicate involving the concerned parameter or variable. This predicate can be deterministic, if for a parameter or variable $a$ it has the form $a = E$ with $E$ free of $a$, or non-deterministic.

Finally, the `then` keyword introduces the list of actions of the event. As all actions are performed simultaneously, a single variable can be modified in at most one action to avoid inconsistencies. There are three kinds of actions, the first one is deterministic and the second and third ones are non-deterministic.

- The simple assignment action $x := E$ replace the occurrences of $x$ by the expression $E$. A special form of this substitution exists for functions: $f(x) := E$ means that the expression $f$ at point $x$, takes the value $E$. This is a shorthand for $f := f \lhd\!\!- \{x \mapsto E\}$, where $\lhd\!\!-$ is the overriding operator.

- The choice by predicate action $x :| P$ arbitrarily chooses a value for the variable $x$, such that the predicate $P$ is satisfied. In $P$, often called *before-after predicate*, the value of the variable before the action is represented by $x$ and the value of the variable after the action is represented by $x'$.

- The choice from set $x :\in S$ arbitrary chooses a value from the set $S$. This is the same as the choice predicate $x :| x' \in S$.

Sometimes, when the set of actions is empty for an event, this set is represented by the *skip* keyword. For notational convenience, multiple single

actions $x := E$, $x := F$ may be grouped into an equivalent multiple action notation $x, y := E, F$.

Every machine has at least one event called `Initialisation`. This event is called once and is used to set the initial values of the variables. It may have only actions and it is the first event called in a machine.

Below is an example of events for the mine pump briefly presented in the previous chapter. A more complete version of this machine will be described in chapter 4.

Listing 3.3: Mine pump machine

**EVENTS**
**Initialisation**
  **begin**
    act1 : $pump, bell, methane, waterLevel := OFF, FALSE, FALSE, LOW$
  **end**
**Event** $high\_water\_detected \mathrel{\widehat{=}}$
  **when**
    grd1 : $waterLevel = HIGH$
    grd2 : $methane = FALSE$
  **then**
    act1 : $pump := ON$
  **end**
**Event** $low\_water\_detected \mathrel{\widehat{=}}$
  **when**
    grd1 : $waterLevel = LOW$
  **then**
    act1 : $pump := OFF$
  **end**
**Event** $updateMethane \mathrel{\widehat{=}}$
  **any**
    $status$
  **where**
    grd2 : $status \in BOOL$
  **then**
    act1 : $methane := status$
  **end**
**Event** $updateWaterLevel \mathrel{\widehat{=}}$
  **any**
    $level$
  **where**
    grd2 : $level \in LEVEL$
  **then**
    act1 : $waterLevel := level$
  **end**
**Event** $methane\_detected \mathrel{\widehat{=}}$
  **when**
    grd1 : $methane = TRUE$
  **then**
    act1 : $pump := OFF$
  **end**
**END**

A summary of Event-B expressions notations can be found in [Robinson, 2009].

### 3.2.3 Proof Obligation Rules

Proof obligations define what has to be proved in an Event-B model. There are eleven kinds of proof obligation rules that can be generated from a model, and partially or totally proved by automatic tools like RODIN [RODIN, 2010].

A complete description of the proof obligations can be found in [Abrial, 2010], the most interesting ones are presented hereafter. Since all actions can be represented as a choice by predicate action with a before-after predicate, for what follows, all actions are normalized under this form. To explain the proof obligations, we will consider a machine $M(s, c, v)$ where $s$ denotes the seen sets, $c$ the seen constants and $v$ the variables of the machine. $Ax(s, c)$ represents the seen axioms and theorems and $Inv(s, c, v)$ represents the local invariants and theorems.

**Invariant Preservation** The invariant preservation proof obligation rule ensures that each invariant of a machine is preserved by each event.

For each invariant $inv$ and event $evt$ with a guard $Guard(s, c, v, x)$, where $x$ represents the parameters of the event, and a before-after predicate $BAP(s, c, v, x, v')$ where $v'$ is the values of the variables after the event, and invariant $inv(s, c, v)$, we will have to prove that:

| | |
|---|---|
| Axioms and theorems | $Ax(s, c)$ |
| Invariants and theorems | $Inv(s, c, v)$ |
| Guards of the event | $Guard(s, c, v, x)$ |
| Before-after predicate of the event | $BAP(s, c, v, x, v')$ |
| $\vdash$ | $\vdash$ |
| Modified Specific Invariant | $inv(s, c, v')$ |

Modified Specific Invariant represents the considered invariant $inv$ with updated variables $v'$.

**Feasibility** This proof obligation rule ensures that for variables $v$ of a machine, non-deterministic actions are feasible, *i.e.* the before-after predicates of those actions are declared in such a way that a value can effectively be found for the variables $v$. For each event $evt$ and each before-after predicate $BAP(s, c, v, x, v')$ of a non-deterministic action $act$, we will have to prove that:

| | |
|---|---|
| Axioms and theorems | $Ax(s, c)$ |
| Invariants and theorems | $Inv(s, c, v)$ |
| Guards of the event | $Guard(s, c, v, x)$ |
| $\vdash$ | $\vdash$ |
| $\exists v' \cdot$ Before-after predicate | $\exists v' \cdot BAP(s, c, v, x, v')$ |

**Guard Strengthening** To be sure that when a concrete event is enabled, then so is the corresponding abstract one, this proof obligation guarantees that the concrete guard of the concrete event is stronger than the abstract one in the abstract event.

Witnesses are used in refining events to assign a value to parameters and variables that have disappeared during the refinement. The concrete event is thus made more precise than the abstract one since if parameters and variables has disappear, it means that their values has been fixed to a constant. For each concrete event $evt$ with a witness predicate $Wit(x, y, s, c, w)$, where $y$ represents the parameters of the abstract event and $w$ the variables of the abstract machine, refining an abstract event $evt_{abs}$ with an abstract guard $Guard_{abs}(s, c, w, y)$, we will have to prove that:

| | |
|---|---|
| Axioms and theorems | $Ax(s, c)$ |
| Abstract invariants and theorems | $Inv_{abs}(s, c, w)$ |
| Concrete invariants and theorems | $Inv(s, c, v, w)$ |
| Concrete event guards | $Guard(s, c, v, x)$ |
| Witness predicates for parameters | $Wit(x, y, s, c, w)$ |
| $\vdash$ | $\vdash$ |
| Abstract event specific guard | $Guard_{abs}(s, c, w, y)$ |

The abstract invariants and theorems are represented by $Inv_{abs}(s, c, w)$ and the concrete invariants and theorems are represented by $Inv(s, c, v, w)$, with variables $v$ declared in the concrete machine and seeing the abstract variables $w$, coming from the abstract machine, in case of a glueing invariant for instance.

**Guard Merging** In the same way, when a concrete event is merging two abstract events, this proof obligation ensure that the guard of the concrete event is stronger than the disjunction of the guards of the abstract events. For each event $evt$ refining two abstract events $evt1_{abs}$ with a guard $Guard1_{abs}(s, c, w, y)$ and $evt2_{abs}$ with a guard $Guard2_{abs}(s, c, w, y)$, we will have to prove that:

| | |
|---|---|
| Axioms and theorems | $Ax(s, c)$ |
| Abstract invariants and theorems | $Inv_{abs}(s, c, w)$ |
| Concrete event guards | $Guard(s, c, v, x)$ |
| $\vdash$ | $\vdash$ |
| Disjunction of abstract guards | $Guard1_{abs}(s, c, w, y)$ |
| | $\vee Guard2_{abs}(s, c, w, y)$ |

**Simulation Proof** The guard strengthening proof obligations ensure that when a concrete event is enabled, so is the abstract one. It does not care about the actions of the events. The simulation proof obligation is concerned with this point, by verifying that in a concrete event, the actions are

correct simulations of the actions of the abstract event. Correct simulation meaning that what the concrete event does is not contradictory with what the abstract event does. For each concrete event $evt$ refining an abstract event $evt_{abs}$ and each abstract action $act_{abs}$ with a before-after predicate $BAP_{abs}(s, c, w, y, w')$, we will have to prove that:

| | |
|---|---|
| Axioms and theorems | $Ax(s, c)$ |
| Abstract invariants and theorems | $Inv_{abs}(s, c, w)$ |
| Concrete invariants and theorems | $Inv(s, c, v, w)$ |
| Concrete event guards | $Guard(s, c, v, x)$ |
| Witness predicate for parameters | $Wit(y, s, c, v, x, v')$ |
| Witness predicate for variables | $Wit(w', s, c, v, x, v')$ |
| Concrete before-after predicate | $BAP(s, c, v, x, v')$ |
| $\vdash$ | $\vdash$ |
| Abstract before-after predicate | $BAP_{abs}(s, c, w, y, w')$ |

**Non-deterministic Witness** Witnesses are used in refining events to assign a value to parameters and variables that have disappeared during the refinement. This proof obligation rule ensures, for a parameter or variable $x$, that this value really exists regarding the witness predicate. For each concrete event $evt$ and each abstract parameter $y$, we will have to prove that:

| | |
|---|---|
| Axioms and theorems | $Ax(s, c)$ |
| Abstract Invariants and theorems | $Inv_{abs}(s, c, w)$ |
| Concrete Invariants and theorems | $Inv(s, c, v, w)$ |
| Concrete event guards | $Guard(s, c, v, x)$ |
| $\vdash$ | $\vdash$ |
| $\exists y \cdot$ Witness | $\exists y \cdot Wit(y, s, c, v, x)$ |

**Theorem Proof** Theorems are usually used to simplify proofs and make automatic proof obligations resolutions easier. This last proof obligation rule is concerned with the proof of those theorems in the different machines and contexts of the model.

### 3.2.4 Event-B Model Decomposition Techniques

The traditional Event-B approach is for now quite linear and not very modular. Works are currently ongoing to solve those problems [Abrial, 2009, Butler, 2009, Pascal and Silva, 2009] by decomposing an Event-B model. Decomposition makes it possible to manage the complexity of models that increases through the refinement process.

Pascal and Silva present in [Pascal and Silva, 2009] a description of the two techniques used to split a machine into smaller pieces. The first one,

called *Event-Based decomposition* or B-style decomposition encapsulates the variables in different machines together with the events or parts of events that concern those variables. A variable will thus not appear in more than one machine. The events that have been split will need to be synchronized in order to ensure the functionalities of the original machine. The synchronization will take place by an exchange of inputs and outputs between the synchronized machine's events. Figure 3.6 shows how a machine $M$ is split into two machines $M_1$ and $M_2$ with a shared event $e_2$. The Event-Based decomposition will not be explored further here and more details about this decomposition can be found in [Butler, 2009].

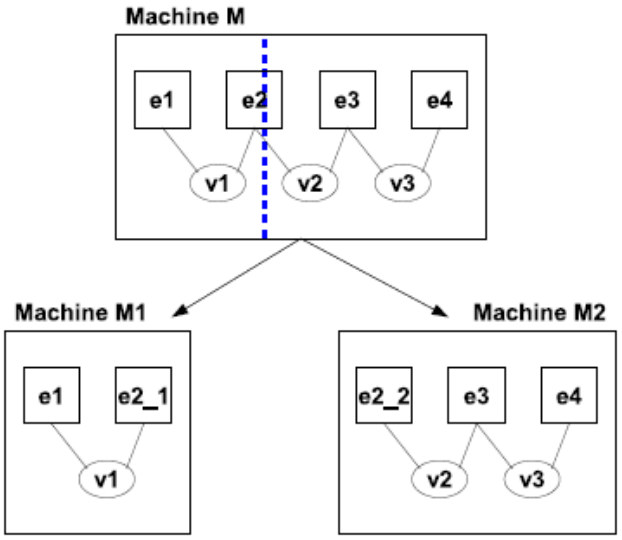


Figure 3.6: Event-Based decomposition [Pascal and Silva, 2009]

The second technique, called *State-Based decomposition* or A-style decomposition [Pascal and Silva, 2009] splits the variables in different machines. A variable may thus be present in more than one machine. Such a variable is called *shared variables*. One of the machines will be the one which effectively updates a shared variable. To keep the other machines synchronised, a special event, called *external event*, will be added to those other machines.

Shared variables must be kept synchronized between the different machines if they are refined. A simple way to overcome this is to forbid data refinement. Data refinement takes place when a variable is refined in a submachine using a glueing invariant. As proved by Abrial in [Abrial, 2009], the system can be rebuilt into a single machine at the end of the process. In practice this will rarely be done since the different machines will lead to different software components.

### State-Based Decomposition

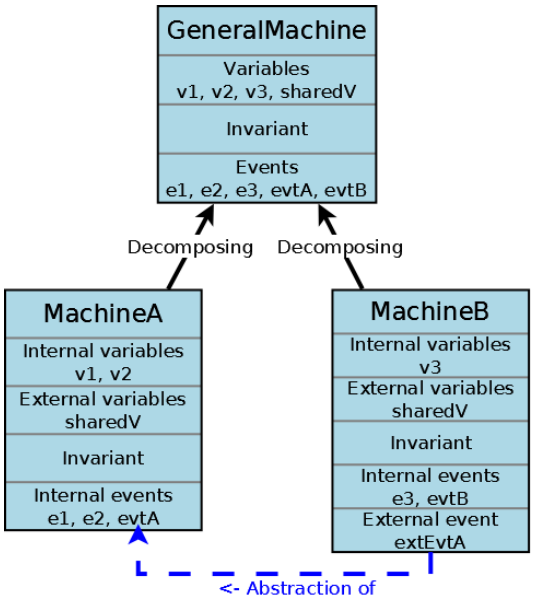


Figure 3.7: Decomposition of a general machine into two sub-machines

In the State-Based decomposition, for a general machine, variables and events will be distributed to several machines with some of those variables present in more than one machine decomposing the general machine. It is important to notice here that the machines are not refining the general machine, but are *decomposing* it. When a machine $B$ refines another machine $A$, it means that $B$ is more concrete than $A$ and that the proof obligations are fulfilled. In the case of decomposition, the variables, events and invariants coming from the decomposed machine, are simply copy-pasted in the decompositions, *i.e.* the decomposing machines. In the machines, a distinction is made between the *internal variables* used only in a particular machine and the *shared variables* used in more than one machine.

Figure 3.7 shows an example of decomposition, a machine `A` has an event `evtA` called *internal event* that will modify the value of a shared variable and another machine `B` has an internal event `evtB` using the variable's value in its guard. To express the fact that the variable is not a constant in `B`, an external event `evtExtA` will be added to `B` corresponding to an abstraction of the internal event `evtA` in `A`. The added event `evtExtA` is present in `B` to synchronize the update of the shared variable in the general machine between machine `A` and machine `B`. As when an abstract event is refined by a concrete event, triggering the concrete event `evtA` implies that its abstraction `extEvtA` is also triggered.

It is clear that shared variables coming from the general machine may be

replicated in more than one machine decomposing it. The problem is that each machine could normally refine its variables and the same replicated variable could be refined in one way in one refinement and in another way in another refinement. If this happens, the two sub-machines can't communicate any longer as they are not using the same convention on the shared variable. Such a variable has a special status in the sub-machines where they reside saying that this variable has always to be present in the state space of any refinement of the machine. A shared variable can thus not be data-refined or if it is, the variable has to be refined in the same way in each sub-model using the variable, which can be heavy.

The same argument is used with external events. Those events are present to notify that a shared variable has not a constant value. This external event will have a concrete implementation in one machine, where this event is an internal event, and will stay abstract in all other machines. As for shared variables, an external event may not be refined.

### 3.2.5 Event-B's Supporting Tool

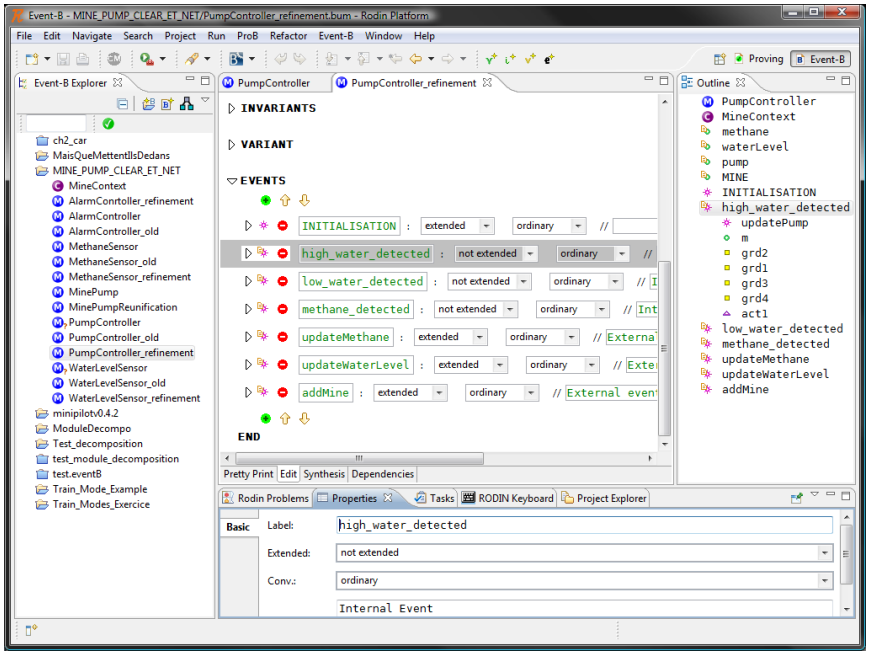


Figure 3.8: RODIN editor: print screen of the mine pump example [RODIN, 2010]

RODIN [RODIN, 2010] is a tool that supports the Event-B method. It is based on the Eclipse platform and includes an Event-B machine/context editor and a proof obligation tool. The proof obligation tool generates the

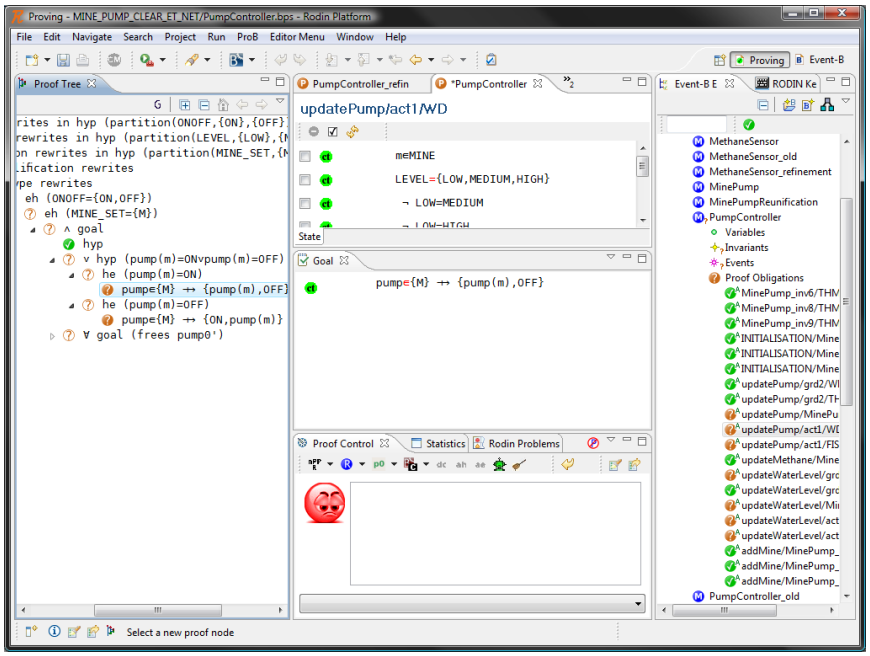


Figure 3.9: RODIN proof obligations tool: print screen of the mine pump example [RODIN, 2010]

proof obligations for the different elements of the Event-B model and helps to automate the proofs as much as possible.

Figure 3.8 shows a print screen of the mine pump example encoded with RODIN. One the left side, the project explorer shows the different Event-B projects with the machines and contexts into each one. On the right side, an explorer shows the different elements of the currently selected machine or context. On the center of the screen, the machine/context editor allows to edit the currently opened machine/context. On the bottom of the screen, the property editor allows to edit the properties of the opened machine/context.

Figure 3.9 shows a print screen of RODIN's proof obligations tool. On the right side, the Event-B explorer shows the different proof obligations that has been proved in green or that has to be proved in brown. On the left side, the proof obligation tree shows the different steps of the currently opened proof. On the center of the figure, the goal frame shows what has to be actually proved at the currently selected node of the proof obligation tree. The frame on the top shows the different hypothesis taken to reach the selected node of the proof obligation tree. At the bottom of the frame, the proof control view contains the buttons which can be used to perform an interactive proof, *e.g.* adding a hypothesis or using an auto-prover.

### 3.2.6 Requirements Engineering and Event-B

Event-B is used to model complete systems. A general development process could be the creation of an Event-B model from requirement documents. This model will be refined until this specification is fine-grained enough to generate B, using the RODIN2B tool for instance. The generated B specification will then be refined in turn to make automatic code generation possible, providing programs formally derived from an Event-B model. The problem here is the gap between the requirements and the initial Event-B model. Requirements will be translated into Event-B. This translation is usually non-systematic and non-repeatable and relies principally on the skills of the analyst. If we don't deny that the talent of the person in charge of the initial Event-B specification is important, it is perhaps possible to facilitate his job by bringing some methodological perspective in the process. Such attempts actually exist and we will briefly present here some of them.

Abrial describes in [Abrial, 2010] a parallel between requirements expressed in a requirements document and definitions and theorems as they can be found in mathematical books. According to him, requirements documents should, as in those books, separate explanatory text from reference text, which will constitute the requirements as they will be used latter in the development lifecycle. The reference text is constituted by a set of short statements written using natural language. Each one of them has an associate number, for traceability purposes, and a label qualifying the nature of the requirement, *e.g.* FUN for functional requirement, ENV for environment requirement, SAF for safety requirement, *etc.*

Siemens uses a similar method, described in [Falampin et al., 2009]. System requirement specifications correspond to documents written in natural language. An Event-B model is manually derived from those documents, using a refinement plan. This plan's purpose is to help modelling and proof by describing the modelling choices and an abstraction ordering of the requirements. As underlined by [Falampin et al., 2009], the main properties, *e.g.* avoiding collisions in a train transportation system, are usually not explicitly explained, but all the functionalities of the system will be means to reach those properties. This implies that an additional abstraction work is needed when the refinement plan is written.

Other approaches, like the one proposed by Bosch [Lecomte, 2009] that uses Michael Jackson's problem frames, exists. Chapter 5 will present three of those approaches. They use KAOS and its linear temporal logic formal layer to express requirements and then translate them to Event-B models.

# Chapter 4

# KAOS to Event B: Proposed Approach

This chapter presents our approach to construct an Event-B model starting from requirements expressed in a KAOS model. A number of techniques to translate a goal requirement model into an Event-B model already exist. Those will be exposed and compared to our approach in chapter 5.

## 4.1 Presentation of the Approach

We propose in our work a semi-formal method to build a bridge between a KAOS model and an Event-B model. To build this method, we start with the following general objectives:

- Method will work from KAOS to Event-B
- A fine grained traceability should be provided
- Method should be automated when possible
- Iterative/incremental development should be possible
- Method should respect KAOS and Event-B semantics
- Method should be at least semi-formal

With those points in mind, we define our method, where starting from the requirements expressed in a KAOS model, we will build step by step an Event-B model where each element will be justified by a requirement. This justification will be implemented through *traceability links* between the two models. A set of rules will be defined to keep the links between the models consistent. By working so, the KAOS model may be incomplete and enriched later, even if the elaboration of the Event-B model has started. The construction process may thus be *iterative* and *incremental*. The analyst can

switch between the two models and modify them, as long as the *traceability rules* are respected.

### 4.1.1 Overview

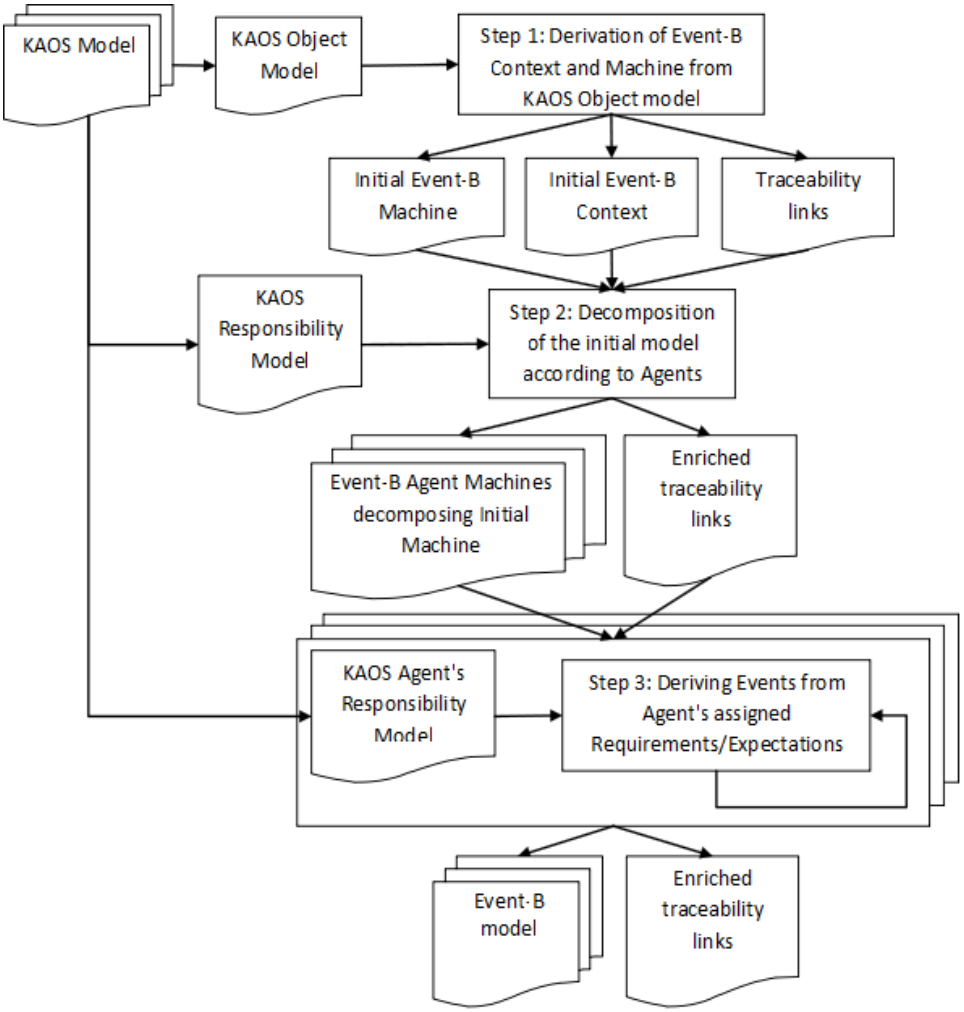


Figure 4.1: Proposed method overview

Figure 4.1 presents an overview of our method. The KAOS model is constructed using different views, leading to different kinds of models linked together:

- In the **first step**, starting from the KAOS object model, an *initial Event-B machine* and an *initial Event-B context* are created to represent the data and very general *update events* are declared in the initial machine to represent the fact that those data evolve in time.

- In the **second step**, the initial machine is decomposed using a special mechanism called state base decomposition explained in subsection 4.3.1, so that for each agent in the KAOS model, exists a machine in the Event-B model associated to this agent. Let us call such a machine an *agent machine*.

  This agent machine will get, during the decomposition process, some variables and events coming from the initial machine representing elements of the KAOS object model. Those variables and events will correspond to the elements of the KAOS object model that the KAOS agent monitors or controls.

- The **third step** is, for each agent machine, to derive in a sub-machine refining the agent machine, the events from the requirements/expectations assigned to the KAOS agent associated to the agent machine. The requirements/expectations are coming from the KAOS agent's responsibility model. As explained in section 2.2.2, a KAOS agent will exert an adequate control on the system items to reach its assigned goals. The system items correspond to the different elements of the KAOS object model, and the adequate control on those items, described on the requirements/expectations, is made explicit through the control and monitor links declared in the KAOS agent's responsibility model.

  This last step is not automatic in our method and will need the skills of the analyst. The goal here was not to automatically derive a complete Event-B model from the KAOS model but rather giving a frame and directions to derive Event-B from requirements, with traceability links to justify and explain the elements of the Event-B model by elements coming from the requirements.

### 4.1.2 Final Result

Figure 4.2 presents how the Event-B model created by the process will be structured. To make the model more readable, the `sees` links between each agent machine and their sub-machines and the initial context are not represented.

Applying the process to a KAOS model with an object model $O$ and $\{a_1, ..., a_n\}$ agents will result in an Event-B model with: an `initial machine` representing the data manipulated by the system coming from $O$ with general update events to represent the fact that those data evolve; an `initial context` describing the data types used in $O$; a set of agent machines $\{\texttt{AM}_1, ..., \texttt{AM}_n\}$ where each agent machine corresponds to a KAOS agent.

Those agent machines will decompose the `initial machine`. An update event coming from he initial machine will be an internal event, that may be refined in sub-machines, in an agent machine if the agent controls the KAOS element from which the update event comes from. An update event

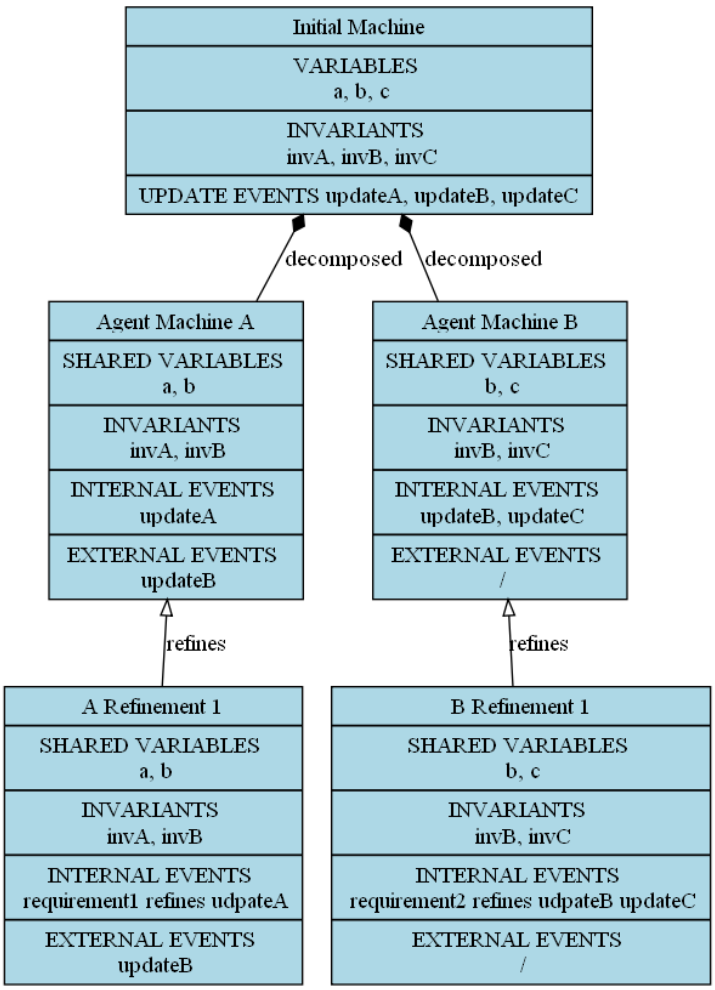


Figure 4.2: Final result of the proposed method

will be an external event, that may not be refined in sub-machines, in an agent machine if the KAOS agent monitors the KAOS element from which the update event comes from.

For each agent machine `AM` corresponding to a KAOS agent $a$, a refinement `AMRef` is created. The requirements/expectations under the responsibility of $a$ will be implemented in `AMRef` by variables, invariants and/or events. If a requirement/expectation updates an element of the KAOS object model, all the events implementing this requirement/expectation will refine the update event corresponding to that KAOS element. This update event must be external in the agent machine and thus correspond to a KAOS element controlled by $a$.

Each time an element of the Event-B model is derived from an element of the KAOS model, a traceability link is recorded to glue the two models. To

preserve consistency between the two models, those links will have to follow some rules described hereafter.

The derivation of the initial machine and context is presented in section 4.2. The decomposition into agent machines is described in section 4.3. Section 4.6 presents the traceability links between the KAOS model and the Event-B model with a list of criteria to keep the links between the two models consistent. In section 4.7, some examples describe what will happen if one model is modified.

### 4.1.3 Example

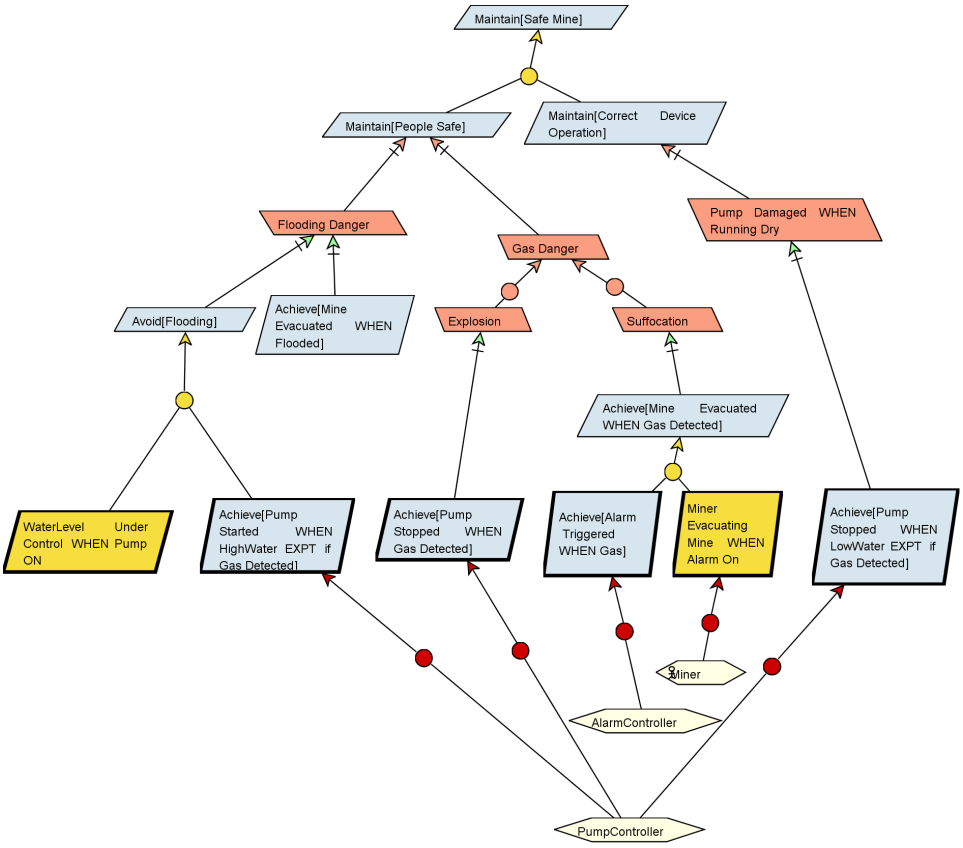


Figure 4.3: Mine pump goal model

In the remainder of this chapter, we will use the mine pump example introduced in chapter 2 to illustrate the different steps of the proposed method.

Figure 4.3 presents the goal model and the different agents responsible for the requirements and expectations. Figure 4.4 shows a view of the responsibility model with controlled and monitored objects: the *PumpController* controls the *pump* attribute and monitors the *methane* and *waterLevel* at-

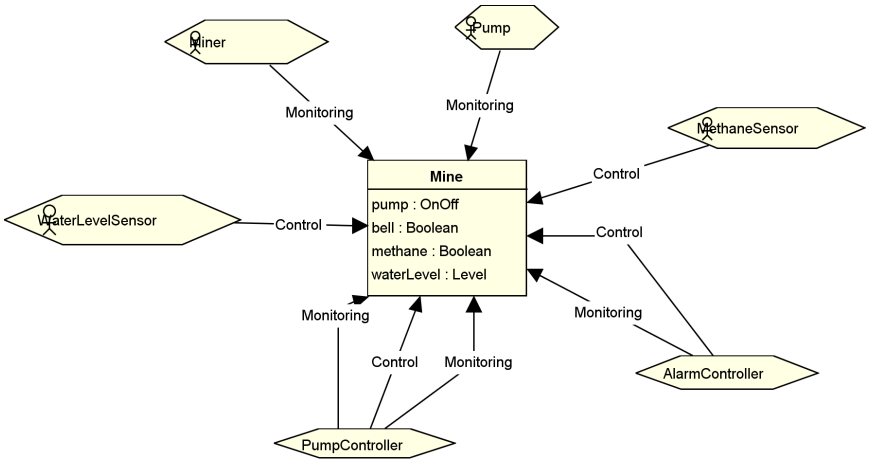


Figure 4.4: Mine pump responsibility model

tributes, the *AlarmController* controls the *bell* attribute and monitors the *methane* attribute, the *WaterLevelSensor* controls the *waterLevelAttribute*, the *MethaneSensor* controls the *methane* attribute and the *Miner* monitors the *bell* attribute.

## 4.2 Step 1: Derivation of Event-B Context and Machine from KAOS Object Model

In KAOS, every concept used in a definition in the goal model has to be defined in the object model. This means that when the goal model is complete, all predicates used in the formal definition of goals and in particular requirements and expectations have been defined in the object model [van Lamsweerde, 2009,Landtsheer, 2007b]. It seems thus interesting to translate in a way or another the object model to Event-B, so concepts manipulated in formulas have an equivalent in the Event-B model.

As Event-B uses set theory to define and manipulate data, the KAOS object model could be quite easily transformed into an *Entity-Relationship-Attribute* model (*ERA*). Tools like DB-Main [REVER, 901] can automatically transform such a model into a relational model compliant with relational databases. The relational nature of the diagram allows getting an Event-B model from it with a simple syntactic transformation. Moreover, as relational databases are the most used database management systems, the relational diagram could be used to generate SQL data definition code. But, this method implies more than one transformation and the generated data definition in the Event-B context and machine may be more difficult to manipulate.

Snook *et al.* define in [Snook and Butler, 2006, yah Said et al., 2009]

a method to transform a UML Class diagram into a classical B machine. This method may be adapted, in fact it can almost be applied as-is, to transform the KAOS Object model which corresponds to a simplified UML Class diagram to an Event-B machine and its associated context.

From now on we will take the following conventions: the name of the *KAOS model elements* will be those defined in the KAOS meta-model [van Lamsweerde, 2009] which corresponds to the concepts presented in section 2.3 on page 11.

### 4.2.1 Object Types and Attributes

A set `OBJECT_SET` of all possible objects belonging to a certain *object* type is defined in the initial context for each object type. The set `OBJECT` of all the existing instances known by the system of a certain object type is defined in the initial machine, that will see the initial context, and belongs to the powerset denoted $\mathbb{P}$ of `OBJECT_SET`:

$$OBJECT \in \mathbb{P}(OBJECT\_SET)$$

The domains of the *attributes* are defined in the initial context. In particular, non standard types or enumerated domains are specified in comprehension or in extension. Attributes are represented in the Machine by a partial or total function according to the *multiplicity* of the attribute, from an element of the `OBJECT` set to an element of the domain of the attribute. The table 4.1 gives the transformation rules for the different multiplicities of an attribute of object type `O`. In this table, $\mathbb{P}_1(TYPE)$ represents the non-empty subsets of $TYPE$, it is equivalent to $\mathbb{P}(TYPE) \setminus \{\varnothing\}$.

Table 4.1: Transformation rules for KAOS Attributes

| **KAOS attribute** | **Corresponding function** | **Event-B Invariant** |
|---|---|---|
| a : `TYPE` [1..1] | Total function to `TYPE` | $a \in O \rightarrow TYPE$ |
| a : `TYPE` [0..1] | Partial function to `TYPE` | $a \in O \nrightarrow TYPE$ |
| a : `TYPE` [1..n] | Total function to non-empty subset of `TYPE` | $a \in O \rightarrow \mathbb{P}_1(TYPE)$ |
| a : `TYPE` [0..n] | Total function to subsets of `TYPE` | $a \in O \rightarrow \mathbb{P}(TYPE)$ |

### 4.2.2 Associations and Specializations

**Association** They may be directed or not and will be represented in the initial machine by functions. Table 4.2 on page 57 gives the transformation rules for the different kinds of *directed associations*. An *undirected association* corresponds to two opposite directed associations and can be managed

as two directed associations with an additional invariant saying that if on exists, then the other exists too. For an association linking A to B with multiplicities [a1..a2] and [b1..b2]

A —a1..a2———————————————b1..b2—B

The result in Event-B will be :

A set AtoB according to the rules in table 4.2
A set BtoA according to the rules in table 4.2
An additional invariant:
$$\forall x, y \cdot (x \in A \wedge y \in B) \Leftrightarrow (AtoB(x) = y \Leftrightarrow BtoA(y) = x)$$

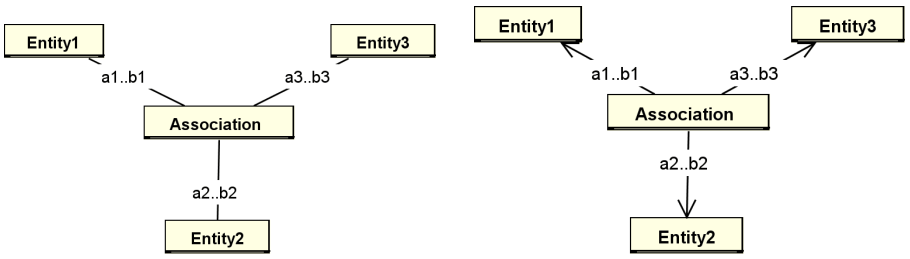


Figure 4.5: N-Ary Association are seen as an Entity with N directed Associations

As shown in figure 4.5, an *n-ary association* will be seen as an *entity* with N directed associations to the different objects of the n-ary association.

**Specialization** In case of *specialization*, instances usually belong to one and only one sub-object type and sub-objects instances are disjoint. As stated by Snook and Butler [Snook and Butler, 2006], when translating from KAOS to Event-B, the instances of the sub-objects will be declared as a subset of super-object's current instances. For instance, three object types, one $Parent$ and two sons $Son1$ and $Son2$ specializing $Parent$ will be translated in Event-B as three variables `PARENT`, `SON1` and `SON2` in the machine and one set `PARENT_SET` in the context. The three fallowing invariants will be added to the machine:

$$PARENT \in \mathbb{P}(PARENT\_SET)$$
$$SON1 \in \mathbb{P}(PARENT)$$
$$SON2 \in \mathbb{P}(PARENT)$$
$$SON1 \cap SON2 = \varnothing$$

The Specialization may be more precise, *e.g.* in case of a *total specialization* where all the instances must be one of a sub-object type then the sub-objects instances sets cover the set of super-object instances and the following invariant is added:

$$SON1 \cup SON2 = PARENT$$

### 4.2.3 General Update Event

In addition to the variables and invariants created to represent the different elements coming from the KAOS object model, the initial machine will also contain general *update events* to represent the fact that those elements may evolve in time. Each element of the KAOS object model will thus be translated by a set of variables, a set of invariants and one update event.

An update event will have actions that update the variables corresponding to the associated KAOS element. In an automated generation, those actions will preserve the invariants generated in this step, but it may be more precise (for now this precision has to be added manually to the Event-B model).

For example, as shown in listing 4.2, the attribute *switch* with a domain $State = \{ON, OFF\}$ will be translated in the initial machine by a variable `switch` with an invariant $switch \in STATE$, where `STATE` is a set defined in the initial context containing the constants `ON` and `OFF`. In place of having an update event `updateSwitch` with an action defined as a before-after predicate saying that $switch' \in STATE$, we may have a more precise action defined as a before-after predicate saying that if $switch = ON$ then $switch' = OFF$ or if $switch = OFF$ then $switch' = ON$. So, not only values at a given state are constrained, but also state transitions.

### 4.2.4 Example: Initial Machine and Context for the Mine Pump

By applying the procedure described in this section to the mine pump example, we get the initial context from listing 4.1 and an initial machine from listing 4.2 describing the objects of the KAOS object model. The initial machine includes the attributes and the update events for all those attributes, note here that in the listing 4.2 only the update method for the pump attribute has been shown. The update method of others attributes follows the same pattern. The complete machines of this example can be found in annex A.

Listing 4.1: Mine pump example: Initial context

```
CONTEXT MineContext
SETS
    ONOFF, LEVEL, MINE_SET
CONSTANTS
    ON, OFF, LOW, MEDIUM, HIGH, M
AXIOMS
```

axm1 : $partition(ONOFF, \{ON\}, \{OFF\})$
axm2 : $partition(LEVEL, \{LOW\}, \{MEDIUM\}, \{HIGH\})$
axm3 : $partition(MINE\_SET, \{M\})$
**END**

Listing 4.2: Mine pump example: Initial machine

**MACHINE** MinePump
**SEES** MineContext
**VARIABLES**
MINE, pump, bell, methane, waterLevel
**INVARIANTS**
inv1 : $MINE \in \mathbb{P}(MINE\_SET)$
inv2 : $pump \in MINE \rightarrow ONOFF$
inv3 : $bell \in MINE \rightarrow BOOL$
inv4 : $methane \in MINE \rightarrow BOOL$
inv5 : $waterLevel \in MINE \rightarrow LEVEL$
**EVENTS**
**Initialisation**
**begin**
act1 : $MINE, pump, bell, methane, waterLevel := \varnothing, \varnothing, \varnothing, \varnothing, \varnothing$
**end**
**Event** *updatePump* $\widehat{=}$
General update event for the pump attribute defined in the KAOS object model
**any**
$m$
**where**
grd1 : $m \in MINE$
grd2 : $pump(m) = ON \lor pump(m) = OFF$
**then**
act1 : $pump :| (pump(m) = OFF \land pump' = (pump \lhd\!\!- \{m \mapsto ON\}))$
$\lor (pump(m) = ON \land pump' = (pump \lhd\!\!- \{m \mapsto OFF\}))$
**end**
**END**

Table 4.2: Transformation rules for KAOS directed Associations [Snook and Butler, 2006]

| |
|---|
| The two object types are A and B and $a1..a2 \rightarrow b1..b2$ in the table represents the multiplicities for an association :<br>A —a1..a2————————b1..b2—> B<br>According to our convention, the objects sets in Event-B will be called `A` and `B`. |
| The *disjoint* macro in the table is defined as:<br>$(\forall a1, a2 \cdot (a1 \in dom(AtoB) \land a2 \in dom(AtoB) \land a1 \neq a2 \Rightarrow AtoB(a1) \cap AtoB(a2) = \varnothing))$ |

| **KAOS association multiplicity** | **Corresponding function** | **Event-B Invariant** |
|---|---|---|
| $0..* \rightarrow 0..1$ | Partial function to `B` | $AtoB \in A \nrightarrow B$ |
| $0..* \rightarrow 1..1$ | Total function to `B` | $AtoB \in A \rightarrow B$ |
| $0..* \rightarrow 0..*$ | Total function to subset of `B` | $AtoB \in A \rightarrow \mathbb{P}(B)$ |
| $0..* \rightarrow 1..*$ | Total function to non-empty subset of `B` | $AtoB \in A \rightarrow \mathbb{P}_1(B)$ |
| $0..1 \rightarrow 0..1$ | Partial injection to `B` | $AtoB \in A \rightarrowtail B$ |
| $0..1 \rightarrow 1..1$ | Total injection to `B` | $AtoB \in A \rightarrowtail B$ |
| $0..1 \rightarrow 0..*$ | Total function to subsets of `B` which don't intersect | $AtoB \in A \rightarrow \mathbb{P}(B) \land disjoint$ |
| $0..1 \rightarrow 1..*$ | Total function to non-empty subsets of `B` which don't intersect | $AtoB \in A \rightarrow \mathbb{P}_1(B) \land disjoint$ |
| $1..* \rightarrow 0..1$ | Partial surjection to `B` | $AtoB \in A \twoheadrightarrow B$ |
| $1..* \rightarrow 1..1$ | Total surjection to `B` | $AtoB \in A \twoheadrightarrow B$ |
| $1..* \rightarrow 0..*$ | Total function to subsets of `B` which cover `B` | $AtoB \in A \rightarrow \mathbb{P}(B) \land \text{union}(\text{ran}(AtoB)) = B$ |
| $1..* \rightarrow 1..*$ | Total function to non-empty subsets of `B` which cover `B` | $AtoB \in A \rightarrow \mathbb{P}_1(B) \land \text{union}(\text{ran}(AtoB)) = B$ |
| $1..1 \rightarrow 0..1$ | Partial bijection to `B` (partial injection defined for all the elements of `B`) | $AtoB \in A \rightarrowtail B \land \forall b \cdot (b \in B \Rightarrow (\exists a \cdot (a \in A \land (a \mapsto b) \in AtoB)))$ |
| $1..1 \rightarrow 1..1$ | Total bijection to `B` | $AtoB \in A \rightarrowtail B$ |
| $1..1 \rightarrow 0..*$ | Total function to subsets of `B` which cover `B` without intersecting | $AtoB \in A \rightarrow \mathbb{P}(B) \land \text{union}(\text{ran}(AtoB)) = B \land disjoint$ |
| $1..1 \rightarrow 0..*$ | Total function to non-empty subsets of `B` which cover `B` without intersecting | $AtoB \in A \rightarrow \mathbb{P}_1(B) \land \text{union}(\text{ran}(AtoB)) = B \land disjoint$ |

## 4.3 Step 2: Decomposition of the Initial Model According to Agents

At the end of the first step of our process, we get an initial context and an initial machine representing the KAOS object model. All elements (Object, Attributes and Associations) of a given KAOS object model will be translated in the initial machine in one ore more variables, one or more invariants and one event representing the update of the element.

Decomposition makes it possible to manage the complexity of models increasing through the refinement process. It may be interesting to have an early decomposition to break an initial machine into smaller pieces pertinent with the KAOS agents. This idea has been inspired by Ball's thesis [Ball, 2008], where the behaviour of TROPOS[1] agents modelling concurrent systems in a distributed environment is transposed into Event-B machines. In our case, this choice is made because the KAOS meta-model says that an association or an attribute can be controlled by one and only one agent [van Lamsweerde, 2009, Landtsheer, 2007b, Letier, 2001]. The idea is thus to have separate machines with the attributes monitored and controlled by the agent. Remind that an attribute or association is controlled by an agent if the agent performs one or more operations that modify the attribute or association value. An attribute or association is monitored by an agent if the attribute or association is an input of one or more operation performed by the agent.

However, the Event-Based decomposition (see sub-section 3.2.4 on page 41) used by Ball in its approach may not be used in our case, because a variable coming from the initial machine will be present in more than one machine decomposing the initial machine since the attribute or association associated to this variable may be monitored by more than one agent. The State-Based Decomposition seems to suit our problem better.

The agent machines will get, during the decomposition process, some variables and events coming from the initial machine representing elements of the KAOS object model. Those variables and events will correspond to the elements of the KAOS object model that the KAOS agent monitors or controls. Let us call "an event in an agent machine updating an element of the object model, that the KAOS agent associated to the machine controls" a *control event*. An event in an agent machine updating an element of the object model, that the KAOS agent associated to the machine monitors will be called a *monitor event*. The monitor events are put in an agent machine to represent the fact that a certain element of the object model, monitored by the KAOS agent, may be modified in time. Control events will be the events effectively triggered by the KAOS agent associated to the machine. Control events will be the only events effectively refined in sub-machines refining the

[1] A goal oriented modelisation language [Bresciani et al., 2004]

agent machine. Both monitor and control events are coming from the initial machine through the decomposition process.

### 4.3.1 State-Based Decomposition Applied to the Initial Machine

We propose to use the State-Based decomposition after the creation of the initial machine and context from the KAOS object model, as presented in section 4.2, with one agent machine per KAOS agent. The reason of this choice is simple, the KAOS meta-model states that an attribute or association cannot be controlled by more than one agent [van Lamsweerde, 2009, Letier, 2001, Landtsheer, 2007b]. So it means that in Event-B, a shared variable will be updated in at most one agent machine, while an external event may be placed with each variable coming from the KAOS object model in zero, one or more other agent machines.

The following algorithm gives the different agents machines decomposing an initial machine `InitM` with an initial context `InitC` according to a given KAOS responsibility model:

- For each KAOS agent *ag*:
  - Create an agent machine `AgM`
  - Declare the `InitC` context as seen by the `AgM` machine
  - For each element *elem* of the KAOS object model monitored but not controlled by *ag*:
    - Copy the variables of `InitM` corresponding to this *elem* in `AgM` and mark those variables as shared
    - Copy the update event of `InitM` corresponding to this *elem* in `AgM` and mark this event as external
  - For each element *elem* of the KAOS object model controlled by *ag*:
    - Copy the variables of `InitM` corresponding to this *elem* in `AgM` and mark those variables as shared
    - Copy the update event of `InitM` corresponding to this *elem* in `AgM` and mark this event as internal
  - For each invariant `Inv` of `InitM`:
    - If `Inv` uses only variables present in `AgM`, *i.e.* variables corresponding to an element of the KAOS object model controlled or monitored by the agent *ag*, then copy `Inv` in `AgM`

Note here that an update event will be replicated as an internal event in one agent machine and to external events in zero, one or more agent machines. At this step, all those events have the same definition. Agent

machines will be refined to make the different internal events more concrete. It means that the update events, and thus the external events defined in other agent machines are indeed abstractions of the concrete internal events defined in one agent machine's refinements.

### 4.3.2 Example: Decomposing the Initial Machine for the Mine Pump

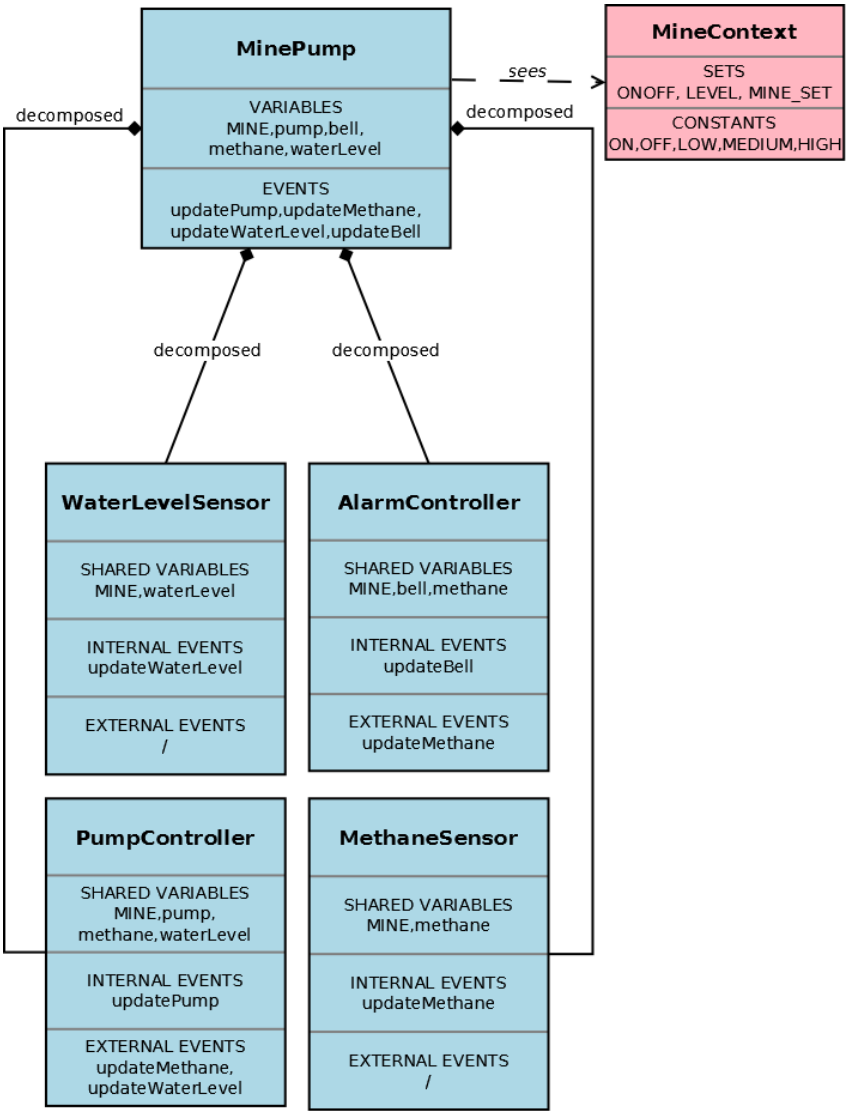


Figure 4.6: Decomposition of the initial machine

The initial machine created for the mine pump will be decomposed into four agent machines. Figure 4.6 presents this decomposition with the agent

machines and their shared variables, external and internal events. For notational convenience, the `sees` link between each agent machine and the `MineContext` has been omitted.

## 4.4 Step 3: Implementing Requirements and Expectations Assigned to an Agent

Each agent machine has now a list of shared variables with invariants related to those variables and a list of events representing the evolution in time of those variables. Those events may be partitioned in *internal events* for the variables linked to KAOS elements controlled by the KAOS agent and *external events* for the KAOS elements monitored by the KAOS agent. Only internal events will be made more concrete by refinement in the agent's sub-machine according to the KAOS agent's behaviour declared in the KAOS requirements/expectations, while external events will be refined in other machines.

One requirement/expectation of the KAOS model will be translated in one or more events, with maybe additional variables and invariants, in the sub-machine refining the agent machine associated to the KAOS agent responsible for the requirement/expectation. If the requirement/expectation needs to update some element of the KAOS object model to be satisfied, assuming that the KAOS agent effectively controls this element in the KAOS model, then the update events associated to that element are refined by the events translating the requirement/expectation, making the update more concrete. This translation is repeated for each requirements or expectations placed under the responsibility of the agent.

To introduce KAOS requirements/expectations for one agent machine, we will proceed as follow: First create a refinement of the agent machine. Every requirement and expectation under the responsibility of the KAOS agent may be translated by zero one or more variable, zero one or more invariants and/or zero one or more events `evts` in this sub-machine. For instance, a requirement/expectation saying that the agent has to keep an error rate value under a certain level may be translated as an invariant in Event-B. Another requirement/expectation saying that the agent has to update a value of the system according to a value coming from the environment may be translated as an event in Event-B.

If the requirement/expectation modifies the value of an element *elem* of the KAOS object model, then the events `evts` implementing the requirement/expectation will refine the update event declared in the agent machine and associated to *elem*. So, every event in the sub-machine updating a variable declared in the parent agent machine will refine the update event that modifies the value of this variable in the parent agent machine. We assume here that the KAOS model is consistent and that the refined events are all

internal events, meaning that the KAOS element they are coming from is effectively controlled by the KAOS agent linked to the refined agent machine.

Listing 4.3 presents the implementation of the requirements under the responsibility of the *PumpController* agent. In this refinement, the `updatePump` internal event is refined in three more concrete events: the `high_water_detected` event implements the requirement *Achieve[Pump Started WHEN HighWater EXPT if Gas Detected]*, the `low_water_detected` event implements the requirement *Achieve[Pump Stopped WHEN LowWater EXPT if Gas Detected]* and the `methane_detected` event implements the requirement *Achieve[Pump Stopped WHEN LowWater Gas Detected]*. This machine and all the other machines of this example can be found in Appendix A.

Listing 4.3: Mine pump example: PumpController_refinement machine

**MACHINE** PumpController_refinement
**REFINES** PumpController
**SEES** MineContext
**VARIABLES**
`methane`, `waterLevel`, `pump`, `MINE`
**EVENTS**
**Initialisation**
*extended*
**begin**
act1 : `MINE`, `pump`, `methane`, `waterLevel` $:= \varnothing, \varnothing, \varnothing, \varnothing$
**end**
**Event** *high_water_detected* $\widehat{=}$
Internal Event derived from requirement Achieve[Pump Started WHEN HighWater EXPT if Gas Detected]
**refines** *updatePump*
**any**
$m$
**where**
grd2 : $m \in MINE$
grd1 : $waterLevel(m) = HIGH$
grd3 : $methane(m) = FALSE$
grd4 : $pump(m) = OFF$
**then**
act1 : $pump(m) := ON$
**end**
**Event** *low_water_detected* $\widehat{=}$
Internal Event derived from requirement Achieve[Pump Stopped WHEN LowWater EXPT if Gas Detected]
**refines** *updatePump*
**any**
$m$
**where**
grd1 : $m \in MINE$
grd2 : $waterLevel(m) = LOW$
grd3 : $pump(m) = ON$
**then**
act1 : $pump(m) := OFF$
**end**
**Event** *methane_detected* $\widehat{=}$
Internal Event derived from requirement Achieve[Pump Stopped WHEN Gas Detected]

**refines** *updatePump*
**any**
$m$
**where**
grd1 : $m \in MINE$
grd3 : $pump(m) = ON$
grd4 : $methane(m) = TRUE$
**then**
act1 : $pump(m) := OFF$
**end**
**Event** *updateMethane* $\widehat{=}$
External event ...
**Event** *updateWaterLevel* $\widehat{=}$
External event ...
**Event** *addMine* $\widehat{=}$
External event ...
**END**

### 4.4.1 Environment Agents and Internal Variables

In our approach, shared variables and external events may not be refined in sub-machines due to the State-Based decomposition. This is not a limitation, because when the KAOS object model evolves, the Event-B model will evolve too, thanks to the traceability links and rules that will be described in section 4.6. Shared variables and external events are the communication convention between the different agents. But it is not forbidden for a particular machine to have internal variables that may be refined and that will not be known by the other machines. This machine may be an agent machine and variables will then represent internal variables of the agent.

As KAOS software agents will be part of the system-to-be, they cannot have internal variables because those variables will be part of the system-to-be too and will thus correspond to an element of the KAOS object model. On the other hand, KAOS environment agents, pre-exist to the system. They have an internal behaviour that is not described in the KAOS model. Only interactions with the system-to-be are described in terms of expectations and controlled or monitored variables. However, it may be interesting to describe some parts of the internal behaviour of KAOS environment agents. To see how the system will react in case of failure of the environment agent for instance. The environment agent machines may have internal variables representing error values, error rates, *etc.*

In our example, the *waterLevel* KAOS attribute is updated by the *WaterLevelSensor* KAOS agent. This agent updates a value of the system-to-be according to a value coming from the environment and not represented in the KAOS model. To test the limits of the model and see how it will react in case of error coming from this sensor, the `WaterLevelSensor` machine may have an internal variable representing the effective water level, corresponding to the real water level according to which the `waterLevel` shared variable is updated. It may also have a variable representing the error rate when the effective water level is measured, *etc.*

## 4.5 Different Kinds of Re-compositions

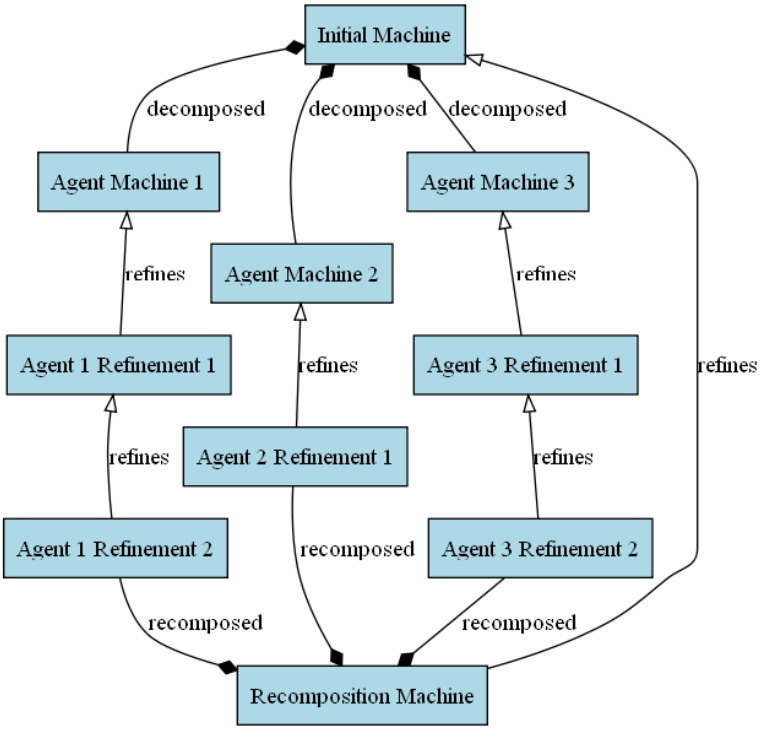


Figure 4.7: Re-composition of the initial machine

After the implementation of the requirements and expectations, we will have different kinds of machines: an initial machine, agent machines decomposing the initial machine, sub-machines refining the agent machines, sub-sub-machines refining those sub-machines, *etc.*

As proved by Abrial in [Abrial, 2009], the system can be rebuilt into a single machine. As shown in figure 4.7, this re-composition will take all the most concrete machines and will rebuild a machine, refining the initial machine where all external events will be replaced by their concrete implementation. This re-composition is done by putting all the variables, invariants and events of the different machines in the re-composed machine and by removing the external events and duplicate shared variables. Note that in case of a refinement chain, like for `Agent 1` in figure 4.7, this chain will be bypassed and the events will directly refine the events of the initial machine. This may be done, thanks to the guard strengthening proof obligations (see section 3.2.3 on page 38), saying that when a concrete event is enabled so is the abstract event.

The re-composition of the different sub-machines can be done for several reasons. The main one is probably the need to observe, via an Event-B

model animator for instance, the behaviour of the whole system. It may act as a kind of checkpoint, used with the client for instance, to see if what he expects to have with the KAOS requirement is actually what he really get from the system, so errors coming from misinterpretations may be detected. Another possibility is to make a partial re-composition, by re-composing all the concrete sub-machines refining a particular set of agent machines. In this case, the external events that are not implemented in one sub-machine will not be removed from the re-composed machine.

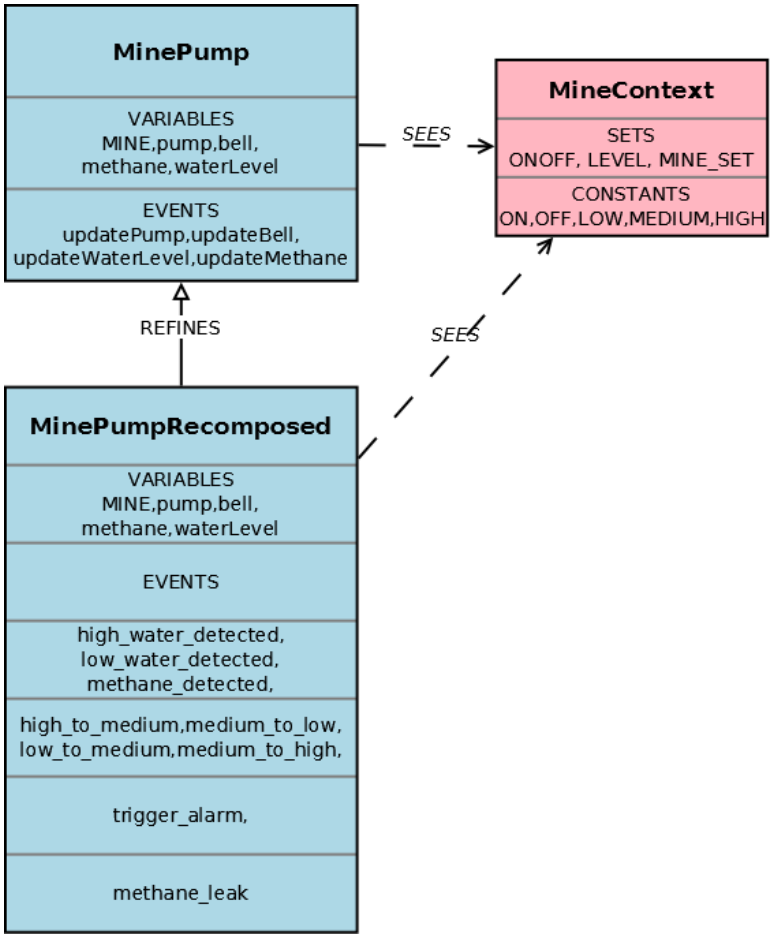


Figure 4.8: Mine pump example: re-composition of the initial machine

For instance, rather than having the behaviour of the all system, we may have the behaviour of the environment by re-composing the sub-machines refining the agent machines associated to a KAOS environment agent. In the same order of idea, we may have the behaviour of the system-to-be by taking the sub-machines refining the agent machines associated to a KAOS software-to-be agent. It may also be useful if we want to observe the interactions of

two particular agents, *etc.*

In our example, the re-composition of the sub-machines refining the agent machines will give a re-composed machine `MinePumpRecomposed`, refining the initial machine `MinePump` with the internal events of the different sub-machines. Figure 4.8 shows a graphical view of this re-composed machine. The complete description of this machine can be found in listing A.11 on page 116 in annex A.

## 4.6 Traceability Between KAOS and Event-B

The idea here is to have rules to justify elements of the Event-B model by elements coming from requirements. The goal is to avoid over-specification and to guaranty that if requirements are discovered or corrected during the elaboration of the Event-B model, the requirements documents will be adapted too to maintain consistency between the specifications and the requirements.

### 4.6.1 Definitions

Before going further, let us recall some definitions coming from chapter 2, used in this section to express rules hereafter :

- An *abstract object* in KAOS is an entity, an agent or an event. Both agent and event may, like in UML, have a "data part" with attributes.
- An *attribute's domain* in KAOS is a domain of values defining the type of an attribute. This domain may be built-in or user defined.
- An *N-Ary association* in KAOS is an association with a multiplicity strictly greater than two.
- An *undirected association* is a bidirectional association.
- An *IsA link* in KAOS is a specialization link taking place between two abstract objects.
- A *domain property* in KAOS is a property guaranteed by the environment. This property is assumed to be always true.

We also clearly define here what are the initial context and machine :

**Definition 4.6.1.** *The initial context is the context derived from the KAOS object model.*

**Definition 4.6.2.** *The initial machine is the machine derived from the KAOS object model with all its variables, invariants and events justified by elements of the KAOS object model.*

During the derivation, *traceability links* are created between the elements of the KAOS model and the elements of the Event-B model. Those links are *derivation links* as defined by van Lamsweerde in his hierarchy [van Lamsweerde, 2009]. A derivation link between two models $A$ and $B$ expresses the fact that *$B$ is build from $A$ under the constraint that $A$ must be satisfied.* In our case, it means that the specification of the system-to-be expressed in Event-B ($B$) has to meet the requirements expressed in KAOS ($A$). This kind of link is vertical in the sense that they take place for a single version of the system, opposed to horizontal links, such as a variant or revision link that take place between different versions.

**Definition 4.6.3** (Traceability Link)**.** *There is a traceability link between one element of the KAOS model and one or more elements of the Event-B model if the Event-B elements are derived from the KAOS element. Traceability links may be defined as a surjective function: traceability* : $B \twoheadrightarrow K$ *where $B$ is the set of Event-B elements belonging to an Event-B model derived from a KAOS model containing the elements in $K$. $K$ contains all the KAOS elements of the object model, all the agents with the requirements/expectations they are responsible for and all the monitor and control links.*

### 4.6.2 Initial Machine and Context

Now that we are done with the vocabulary, let us define rules for the Event-B model derived from the KAOS model. First, we will define rules for the initial machine and context. Those two elements are build by the transformations described in section 4.2. The following rules must be respected to keep the Event-B model consistent with the KAOS model.

#### Initial Context

Here are the rules for the sets, axioms and constants that can be found in the initial context.

**Rule 4.6.1.** *Each carrier set in the initial context must be linked to one abstract object, or one attribute domain or one N-Ary association.*

**Rule 4.6.2.** *Each constant in the initial context must be linked to one attribute's domain.*

**Rule 4.6.3.** *Each axiom in the initial context must be linked to one attribute's domain.*

#### Initial Machine

Here are the rules defined for the invariants, variables and events defined for the update of those variables.

**Rule 4.6.4.** *Each variable in the initial machine must be linked to one abstract object or one attribute or one directed association or one undirected association or one N-Ary association.*

**Rule 4.6.5.** *Each invariant in the initial machine must be linked to one abstract object or one directed association or one undirected association or an IsA link or an N-Ary association or a domain property.*

We will call an *update event* an event corresponding to the update of one KAOS element which can be an abstract object, an attribute, an N-Ary association, a directed association or an undirected association.

**Rule 4.6.6.** *Each event in the initial machine must be an update event and is thus linked to one abstract object or one attribute or one directed association or one undirected association or one N-Ary association.*

Note that one element in KAOS may be translated in more than one variable in Event-B, *e.g.* the undirected association that is transformed into two sets and an additional invariant.

**Rule 4.6.7.** *Each variable in the initial machine must appear in one and only one update event.*

**Rule 4.6.8.** *Each KAOS element which can be an abstract object, an attribute, an N-Ary association, a directed association or an undirected association must appear in the initial machine.*

### 4.6.3 Agent Machines and their Refinements in the Event-B Model

According to our approach, the initial machine will be decomposed in a set of agent machines. Those agent machines will then be refined independently to describe the behaviour of each KAOS agent of the system under study. Here are rules for those agent machines.

#### Machines

An agent machine is a machine decomposing the initial machine. An agent machine is linked to one KAOS agent. A machine is said as *indirectly linked to an agent* if it is refining an agent machine or a machine indirectly linked to an agent. A machine will be said to be *linked to an agent* if it is an agent machine or it is indirectly linked to an agent.

**Rule 4.6.9.** *Each machine in the Event-B model that is not the initial machine must be linked to one agent or must be a re-composition, as defined in section 4.5, of several machines.*

**Events**

An event is said as *linked to a requirement or an expectation* if it is directly linked to a requirement or an expectation or if it is refining an event linked to a requirement or an expectation. Such an event will be part of the implementation of the requirement or the expectation.

**Rule 4.6.10** (Requirements Traceability)**.** *Each internal event in the machines that are not the initial machine or an agent machine must be linked to a requirement or expectation under the responsibility of the agent linked to the machine.*

Note that a recomposed machine is implicitly linked to all the agents corresponding to the machines that are part of the re-composition. We will say that an event $c$ refines another event $a$ ($c$ refines $a$) if it refines it directly or if it refines a third event $b$ that refines the other event $a$ ($c$ refines $b$ refines $a$).

The three following rules are not directly related to traceability links, but are important to keep the Event-B model consistent with respect to the KAOS model. It enforces what has been explained in section 4.4 using the more precise vocabulary introduced at the beginning of this section.

**Rule 4.6.11.** *If an internal event in a machine that is not the initial machine or an agent machine updates the value of variables corresponding to a KAOS element, which can be an abstract object, an attribute, an N-Ary association, a directed association or an undirected association then the event must refine directly or indirectly the update event defined in the agent machine corresponding to this KAOS element.*

**Rule 4.6.12.** *If an internal event in a machine that is not the initial machine or an agent machine refines an update event, the agent linked to the machine must control the KAOS element linked to this update event, which can be an abstract object, an attribute, an N-Ary association, a directed association or an undirected association in the KAOS model.*

To ensure the KAOS meta-constraint saying that a KAOS element may be controlled by at most one agent (see subsection 2.3.5 on page 21), the following rule is defined:

**Rule 4.6.13.** *Each update event in the initial machine may be defined as an internal event in only one agent machine.*

### 4.6.4 General Rule

The last rule to add is the one that links all the others. We said in the beginning of this section that our goal is to keep the KAOS model consistent

with the Event-B model, so each element of the Event-B model may be explained by element(s) of the KAOS model. Note that we haven't considered the semantic of the generated Event-B model with regard to the semantic of the KAOS model. However, the idea of decomposing the Event-B initial machine into several agent machines, where each agent will give a software component running in parallel with the other components does not seem to contradict the KAOS semantic of a system, defined as the parallelization of the agent's behaviours. This discussion is left as a future work that has to be done.

**Rule 4.6.14** (Models Consistency)**.** *If the rules 4.6.1 to 4.6.13 are respected, then the Event-B model is consistent with the KAOS model it comes from. Elements of the Event-B model are effectively derived from elements of the KAOS model and may be explained by them.*

## 4.7 What happens if ...

One of the advantage of the method we have proposed in this chapter is its non-monolithic characteristic. The KAOS model may be modified and the Event-B model will be adapted without re-generating the complete Event-B model. But going on the other way is also possible, *e.g.* if omissions are detected by the specification, an adaptation of the Event-B model may be transmitted to the KAOS model to keep the requirements consistent with the specifications of the system. This second approach is however less generic than the first one since rules to go from KAOS to Event-B are clearly defined. This section will be limited to repercussions, in the Event-B model, of modifications in the KAOS model by presenting some cases where the Event-B model is update following a modification of the KAOS model.

### 4.7.1 ... an element is added in the KAOS object model?

Adding an element to the KAOS object model will result in a modification of the initial machine and context. This element is added according to the rules described in section 4.2 on page 52 to the initial machine and context and will be propagated to the agent machines and to the sub-machines refining them.

We assume here that this element is not controlled or monitored by a KAOS agent, no external event is thus added to an agent machine or its refinements. Description of what happens when a control link is added is described in section 4.7.5. What happens when a monitor link is added is described in section 4.7.7.

For instance, if we add an mandatory integer attribute *depth* to the *Mine* KAOS object described in subsection 4.1.3, we will update the initial machine, so it has a new variable `depth`. With a new invariant $depth \in$

$MINE \rightarrow \mathbb{N}$ and a new update event `updateDepth` with two parameter $m$ and $l$, two guards $m \in MINE$ and $l \in \mathbb{N}$ and an action $depth(m) := l$. Since the *depth* KAOS attribute is not controlled or monitored by any KAOS agent, the `depth` variable and the `updateDepth` event will not be transmitted to any agent machine.

### 4.7.2 ... an element is removed from the KAOS object model?

When an element is removed from the KAOS object model, the invariants, variables, update event, sets and axioms issued from its translation, according to the rules described in section 4.2 on page 52, in Event-B are removed from the initial machine and context, from the agent machines and from their refinements. Note that before deleting an element, all the control and monitor links will be removed too. We assume that the KAOS meta-constraint, imposing that all the elements used to define goals, requirements and expectations must be defined in the object model, is respected. An element will thus not be removed while at least one requirement or expectation is using it and thus events linked to requirements and expectations will stay correct in the Event-B model.

### 4.7.3 ... an agent is added in the KAOS model?

Adding an agent to the KAOS model means that a new active entity has been identified. The Event-B model will thus be enriched by a new agent machine, decomposing the initial machine. When the agent is added, we assume that it does not monitor or control anything. Those links are added later in the KAOS model.

### 4.7.4 ... an agent is removed from the KAOS model?

Removing an agent from the KAOS model means that an active part of the system is removed. We assume that all the responsibility links between the agent and the requirements/expectations will be removed or moved to other agents before removing it. As an agent may be responsible for a requirement/expectation if and only if he can control all the data that are modified by the requirement/expectation and monitor all the data read by the requirement/expectation, all the monitor and control links will also probably be moved before the deletion of an agent.

If an agent is removed from the KAOS model, the corresponding agent machine and all its sub-machines will be removed from the Event-B model. If one of those machines has been used in a re-composition, all the events coming from the machine will be removed from the decomposition. Pay attention that if the agent was still controlling a piece of data when it is removed and that one of the deleted event in the re-composition was refining

the update event of this piece of data, the general update event coming from the initial machine has to be added in the re-composed machine.

### 4.7.5 ... a control link is added in the KAOS model?

If a control link is added to the KAOS model, the update event linked to the controlled KAOS element will become an internal event in the agent machine corresponding to the KAOS agent. We assume that the KAOS meta-constraint saying that a piece of data can be controlled by one and only one agent (see section 2.3.5 on page 21) is respected.

### 4.7.6 ... a control link is removed from the KAOS model?

In the Event-B model, when a KAOS agent is controlling a KAOS element, it means that the update event of this KAOS element is an internal event in the agent machine linked to the KAOS agent. Removing a control link means that the KAOS agent can no longer modify a certain KAOS element. Deleting a control link may only occur in KAOS when the agent is no longer responsible for requirements/expectations that update the previously controlled element. A more frequent situation will be to move requirements/-expectations responsibilities to another agent and in the same time, move control and monitor links needed to be responsible for those requirements/-expectations to this other agent too.

We assume in what follows that the KAOS agent is no longer responsible for a requirement/expectations that needs the removed control link to be fulfilled. It means that in the sub-machine refining the agent machine, the internal update event had the same definition as in the agent machine before deletion of the control link. If the KAOS agent still monitor the previously controlled KAOS element, then the update event is marked as external in the agent machine and its refinements. If there there is no monitor link between the KAOS agent and the KAOS element, the update event, variables and invariants linked to this element are deleted from the agent machine and its refinements.

### 4.7.7 ... a monitor link is added in the KAOS model?

When a monitor link is added to the KAOS model, it means that a KAOS agent will be notified when a certain KAOS element is updated. In Event-B, it means that the update event linked to this KAOS element is added as an external event in the agent machine linked to the KAOS agent and all its refinements.

### 4.7.8 ... a monitor link is removed from the KAOS model?

If the KAOS element previously monitored by the KAOS agent is not controlled by this KAOS agent, the external update event, variables and invariants linked to the previously monitored KAOS element will be removed from the agent machine and all its refinements.

### 4.7.9 ... a newly created requirement/expectation is assigned to an agent?

If a new requirement/expectation is added to the KAOS model and assigned to an agent, it means that the agent's behaviour is modified. We assume here that the agent has effectively the ability to control and monitor the KAOS elements needed to fulfil the requirement/expectation. The internal and external update events are then already present in the agent machine. If the agent machine has not been refined yet, then the new requirement/expectation will be implemented during the agent machine refinement process with all the other requirement/expectation as described in section 4.4.

If the agent machine has already been refined, it means that all the other requirements/expectations have already been implemented. The new requirement/expectation will be implemented in the first refinement of the agent machine as described in section 4.4 and this implementation will be propagated to all the sub-machines of the agent machine's first refinement.

### 4.7.10 ... a requirement/expectation assigned to an agent is modified?

If a requirement/expectation assigned to an agent is modified in the KAOS model, it means that the agent's behaviour is modified. We assume here that the agent has effectively the ability to control and monitor the KAOS elements needed to fulfil the modified requirement/expectation. The internal and external update events are then already present in the agent machine. The first refinement of the agent machine, where the requirement/expectation is implemented, will be modified according to the new definition of the requirement/expectation. This part is not automatic and will rely on the analyst's skills.

Once the requirement/expectation implementation has been modified in the first refinement of the agent machine, the proof obligations of the sub-machines will be regenerated and will have to be proved correct. Those sub-machines will, in most cases, have to be modified too to fit the new behaviour of the agent.

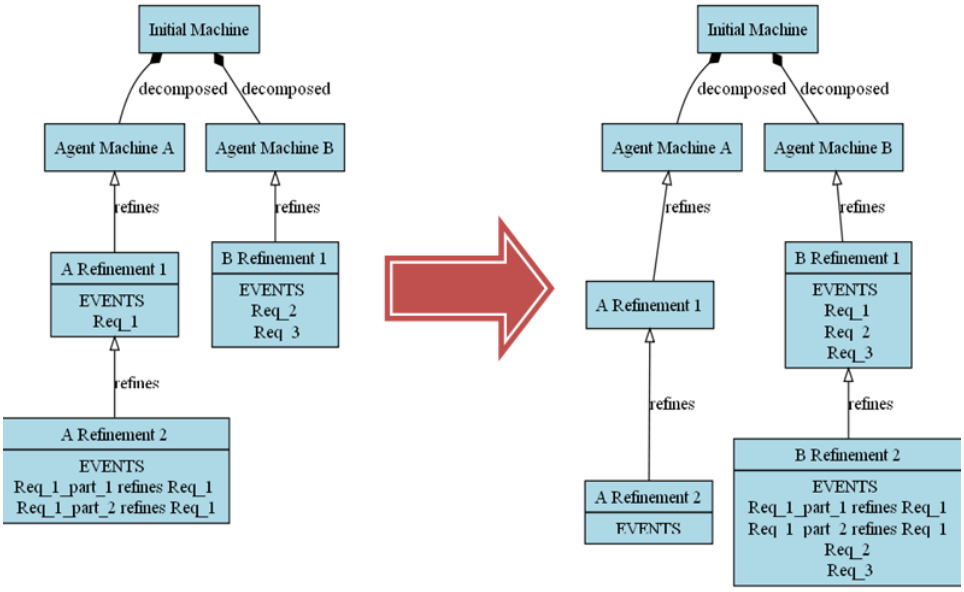


Figure 4.9: Moving responsibility link in Event-B

### 4.7.11 . . . a responsibility links is moved from an agent to another?

A responsibility link in KAOS is translated into an event or an invariant in the machines refining directly or not the agent machine. If it is an event, it refines all the update events corresponding to the data that are modified by the requirement/expectation. The agent has thus the ability to control those data in the KAOS model. Moving a responsibility from an agent to another will thus mean that the implied control links will be moved at the same time.

The events linked to the requirement/expectation are moved from one machine to another. If the event linked to the requirement/expectation has already been refined in sub-machines, the refinements may be moved from the previous agent's "refinement chain" to the new one by completing the actual machines and creating new ones if the new chain is shorter than the previous one. Figure 4.9 shows an example of a refined requirement $Req_1$ moved from the agent $A$ to the agent $B$, where the agent $B$'s machine has not yet been refined. A refinement `B Refinement 2` is created to have the same refinement level as agent $A$'s machine.

## 4.8 A First Implementation

A first implementation of the proposed approach has been elaborated using model to model transformation technologies. This implementation takes on

input a KAOS model and outputs an Event-B model containing the initial machine, the initial context and the agent machines decomposing the initial machine. The implementation of requirements and expectations is not automated here.

This prototype act as some kind of *proof of concept* to show that the proposed approach may be implemented using actual technologies. We will first present those technologies and then discuss the limitations and future works that have to be done.

### 4.8.1 The ATLAS Transformation Language

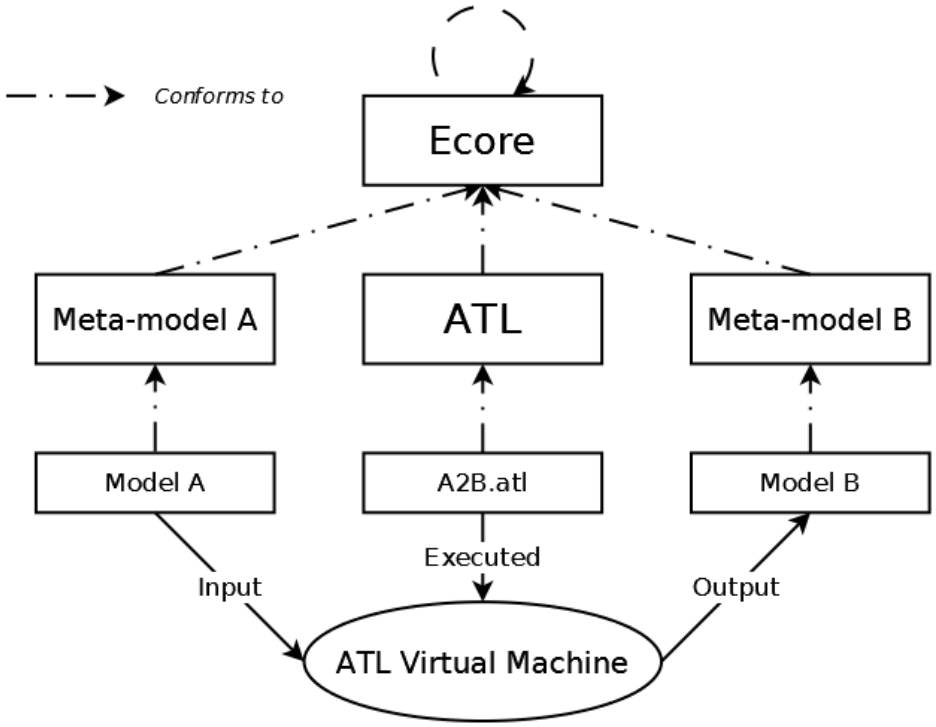


Figure 4.10: ATL transformation structure [Jouault et al., 2008]

The transformation has been implemented in the ATLAS Transformation Language (ATL)[2]. It is a model to model transformation language based on the OMG's QVT specification [OMG, 2007]. It uses both declarative and imperative constructs. Declarative constructs are preferred but imperative ones are left to perform complex transformations easier. An ATL transformation program will correspond to as set of rules defining how source elements are matched to target elements with the initialization of these target

[2]ATL is now part of the Model to Model project (`http://www.eclipse.org/m2m/`) supported by the Eclipse foundation

elements [Jouault et al., 2008].

Figure 4.10 shows how an ATL transformation takes place in a synthetic view. A transformation is an instance of the ATL meta-model and is executed on a specific virtual machine. It is defined to take a `A` meta-model instance as input, a KAOS model conform to the KAOS meta-model in our case, and produce a `B` meta-model instance, an Event-B model conform to the Event-B meta-model in our case. The `A` and `B` meta-models have to be themselves expressed in a formalism conform to meta-model, *Ecore* in this case, which is sometime called meta-meta-model. The KAOS meta-model and the Event-B meta-model are expressed here in `.ecore` files. *Ecore* is briefly presented in the next section.

Listing 4.4: Part of the KAOS to Event-B ATL transformation

```
1  rule EntityRule {
2    from entity : KAOS!Entity
3    to set : SIMPLEEVENTB!CarrierSet(
4                              id <- 'ObjModel_'+entity.name+'
                                   _SET',
5                              name <- entity.name+'_SET',
6                              <...>),
7            variable : SIMPLEEVENTB!Variable( <...> ),
8            invariant : SIMPLEEVENTB!Invariant(<...> ),
9            evt : SIMPLEEVENTB!MachineEvent( <...> ),
10           link : SIMPLEEVENTB!EntityObjectDerivation (
                 <...> )
11   do{ < ... >
12         for(attribute in entity.attributes){
13                 thisModule.createAttribute(attribute,
                       entity);
14         }
15   }
16 }
```

For instance, in listing 4.4 the rule `EntityRule` describes how a KAOS *Entity* will be translated in Event-B. As defined in section 4.2, for each KAOS *Entity* encounter in the source model, the output Event-B model will contain a `set` in the initial context for al the possible entity instances, a `variable` in the machine for the instances recorded in the machine, an `invariant` to type the `variable`, a general update event `evt` updating the `variable` and a traceability `link`. The `do` part is an imperative construct that will create the Event-B elements derived from the attributes of the KAOS *Entity*.

### 4.8.2 Ecore Meta-Model

Both input and output models of the ATL transformation are expressed in `.ecore` files. *Ecore* is a meta-model defined in the Eclipse EMF framework[3] used to describe models [Budinsky et al., 2003]. The EMF framework offers, among other things, tools and support to generate everything needed to build a complete editor for an *Ecore* model, including generation of Java classes to manipulate a model instance, default XMI serialization[4], a user interface generator, *etc.*

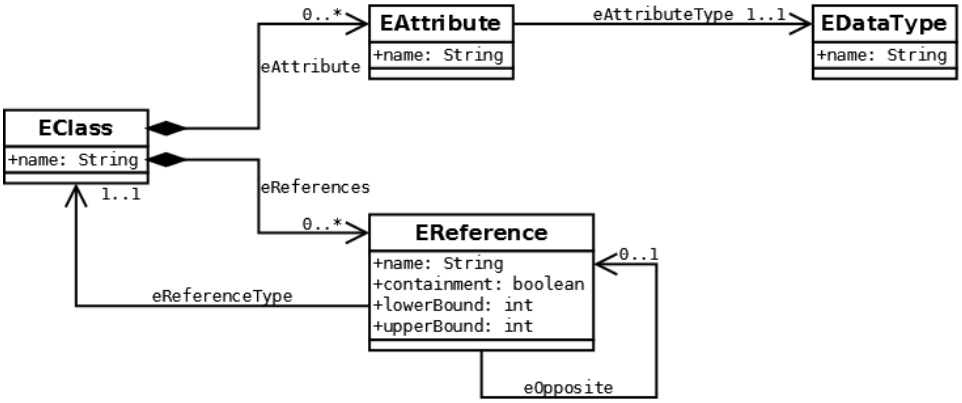


Figure 4.11: Main concepts of the Ecore model [Budinsky et al., 2003]

Figure 4.11 shows the most important part of the Ecore model. The main concepts are the *EClass* that may have one or more super-type, *EAttributes* typed as *EDataType* and may be involved in one or more *Ereferences*. The KAOS and Event-B meta-models used in the prototype are thus expressed in terms of *EClass*, *EAttributes*, *EDataType* and *Ereferences*.

### 4.8.3 Actual State, Limits and Future Implementations

In its actual state, the prototype is limited to the *first* and *second* steps of our approach. The initial context, the initial machine and its decomposition in agent machines are automatically derived from a KAOS model. Step *three*, where agent machines are refined and requirements/expectations are implemented has to be done manually.

Both the prototype and the method have been applied to the mine pump example described in chapter 2 and another example which is actually an exercise, based on the pilot of the Deploy project [Falampin et al., 2009], that has the same proportions than the mine pump example. In this exercise, a device has to managed the different driving mode switching (from fully automated to fully manual) of a train. This other example is not

[3] `http://www.eclipse.org/modeling/emf/`

[4] XML Metadate Interchange defines a way to specify model objects in XML documents

completely described here, but as an indicator, the KAOS model contained height requirements/expectations assigned to three agents and in the generated Event-B model the main agent machine's refinement has six internal events used to manage the modes switchings.

The used meta-model for KAOS is the one defined for Objectiver [Respect-IT, 2009] that may be found in the *kaos-emf* package on the FAUST project's Sourceforge page[5]. This package contains both, the KAOS Ecore meta-model and Java classes to connect a Java application to Objectiver and get the currently edited KAOS model.

RODIN [RODIN, 2010] has actually an EMF plugin with an Ecore meta-model, but since there were compatibility problems between this meta-model and the Eclipse EMF framework, a new meta-model has been defined for Event-B. This new meta-model, called *simpleEventB*, defines basically all the notions that may be found in Event-B, plus the different traceability links between KAOS and Event-B elements. The generated model can not be directly imported in the RODIN tool, and the link between the generated model and the tool has to be implemented.

To illustrate the prototype let us give some numbers: the defined *simpleEventB* meta-model contains 50 EClasses, 27 are used to re-define the Event-B meta-model and 23 are used to define the traceability links; the KAOS meta-model defined in the *kaos-emf* package contains 38 EClasses; the ATL transformation contains 8 rules and is 542 lines long; the execution in Eclipse of the ATL transformation for the mine pump example takes 0,007 seconds; the execution in Eclipse of the ATL transformation for the pilot of the Deploy project example takes 0,009 seconds.

The two main future works for this prototype will be: first the implementation of the traceability checks via the rules described in section 4.6. This can be achieve using the Eclipse Object Constraint Language plugin[6] for instance, which permit to write OCL rules and to evaluate those rules with an EMF model. Secondly, the formal definitions of requirements and expectations will have to be translated automatically into Event-B. This could already partially be done, using De Landtsheer's method described in section 5.3 for formula that use exclusively past time operators.

[5] `http://sourceforge.net/projects/faust/`

[6] OCL - `http://www.eclipse.org/modeling/mdt/?project=ocl`

# Chapter 5

# KAOS to Event B: existing approaches

This chapter presents three existing methods, currently under research for some of them, to derive an Event-B model from a KAOS model. The first one, proposed by Matoussi, works on a KAOS goal diagram, containing "Immediate Achieve" goals and built with milestone-driven and or-refinement patterns. The second approach, proposed by Aziz *et al.* introduces the notion of trigger conditions for events to derive an Event-B model from a KAOS model. The last approach, proposed by De Landtsheer takes linear temporal logic formula expressed exclusively with past operators on input and produces an event-based security policy expressed in Polpa. A syntactic change can translate this Polpa policy to an Event-B model.

The last section of this chapter will present the problems encounter with those approaches in deriving an Event-B model from a KAOS goal model. A comparison between the different approaches of this chapter and our proposed approach presented in chapter 4 will be made at the end of this section.

## 5.1 Expressing KAOS Goal Models with Event-B: A. Matoussi

Matoussi describes in [Matoussi et al., 2008,Matoussi et al., 2009,Matoussi, 2009,Gervais et al., 2009] a process to transform a KAOS goal model into an Event-B specification. This process takes as input a KAOS goal model that is not operationalized and produces an Event-B model corresponding to a specification that satisfies the requirements described in the input model.

This process is based on refinement patterns. The idea is that each refinement pattern used in the KAOS model will correspond to a refinement step in the Event-B model. Actually the process works with functional "Immediate Achieve" goals which are the most commonly used goal type. Those goals have to be formally defined with an assertion of the form $A \Rightarrow \Diamond B$,

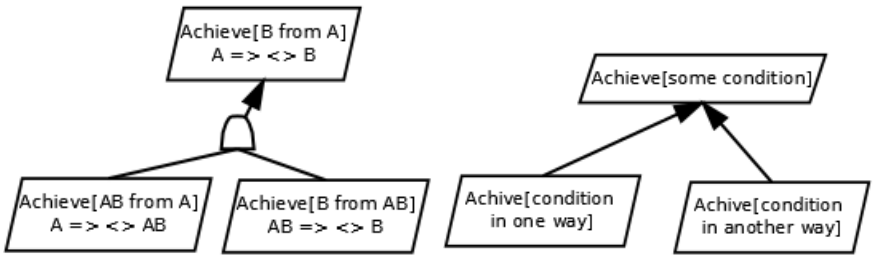


Figure 5.1: Milestone-driven refinement and Or-refinement

which says that from a state where $A$ is true, another state where $B$ is true can be reached someday. The supported patterns, presented in figure 5.1, are the milestone-driven refinement pattern, used when a target condition $B$ can be reached from a current condition $A$ with an intermediate condition $AB$ and the or-refinement pattern, used when a goal can be satisfied in different ways.

The process in figure 5.2 has two phases: the first one creates an Event-B representation of the goal model. The initial Event-B model includes the definition of a context with all the types used for data and the definition of an initial machine. This initial machine represents the root goal $G$ of the KAOS model and each refinement in this model has to follow one of the two patterns described in figure 5.1. Each refinement step in the goal model will correspond to a refinement step of the Event-B machine, so it produces a chain of refined machines where each machine will correspond to a "stage" in the goal model.

The second phase formally derives an Event-B specification that satisfies the requirements expressed in the goal model. To do this, it takes as input the goal model and the Event-B representation of this model created in the first phase. This second phase correspond to the operationalization process that can be performed in KAOS and guaranties that operations preserve all the properties of the goal model. As in the first phase, the initial Event-B model will be defined for the root goal $G$ of the model and each refinement in the goal model following one of the two patterns will correspond to a refinement in the Event-B model.

### 5.1.1 First Phase

Formally speaking, a KAOS goal is seen as a property that the system has to establish:

$$\begin{gathered}\text{Achieve[G]} \\ A \Rightarrow \Diamond B\end{gathered}$$

This property will be represented as an event in the Event-B model where the premise of the implication is transcribed in the initialization event of the

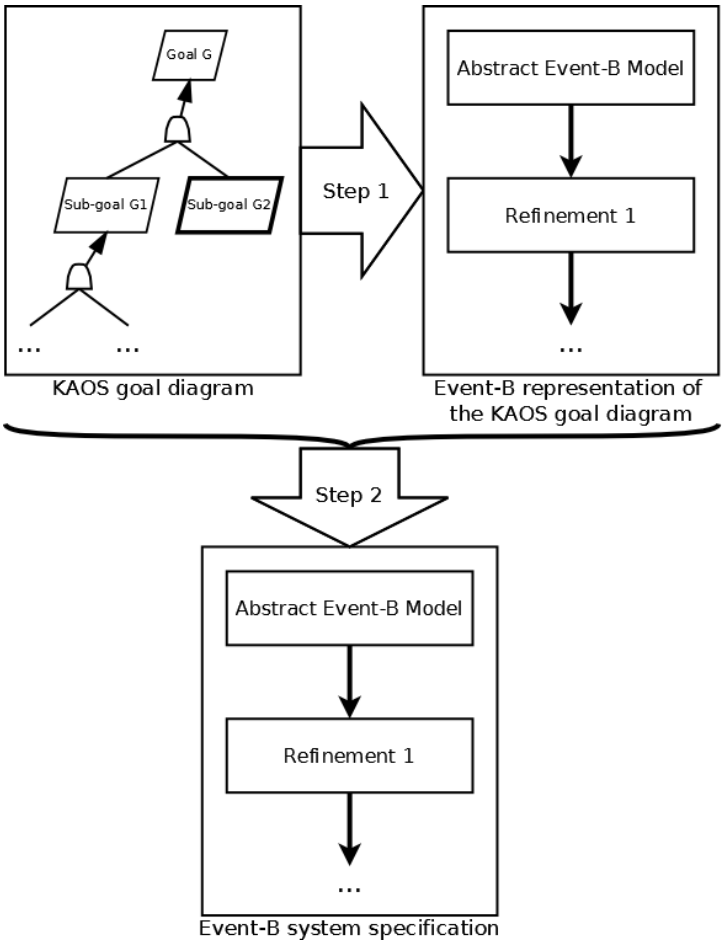


Figure 5.2: Expressing KAOS with Event-B: overview [Gervais et al., 2009]

machine and the consequence of the implication is transcribed in the then part of the event `EvtG` associated to the goal $G$. An execution of this event means that the goal $G$ has been satisfied. The guard of `EvtG` is set to true to express the fact that at this level the goal can always be achieved.

Listing 5.1: KAOS expressed in Event-B: initial machine

```
MACHINE EventBGoalModel_level_0
SEES ModelContext
VARIABLES
    Manipulated data
INVARIANTS
    inv : Data types definitions
EVENTS
Initialisation
    begin
```

```
        act : A
    end
Event EvtG ≙
    where
        grd : TRUE
    then
        act : B
    end
END
```

### Milestone-Driven Refinement

In general, when we have a milestone-driven refinement like the left refinement in figure 5.1, it means that the parent goal $G$ is satisfied when all the sub-goals $\{G_1, ..., G_n\}$ have been satisfied. In figure 5.1 and in the rest of this example, $n$ is limited to 2 to avoid complicate and useless explanations. The `EvtG` event of the parent machine is refined into a new event `EvtG` taking as pre-condition the conjunction of the functional post-conditions of the children, $G1$ and $G2$ in this case. The refinement of goal $G$ following the pattern described in figure 5.1 will give the machine represented in listing 5.2.

Listing 5.2: KAOS expressed in Event-B: milestone refinement machine

```
MACHINE EventBGoalModel_level_1
REFINES EventBGoalModel_level_0
SEES ModelContext
VARIABLES
    Manipulated data
INVARIANTS
    inv : Data types definitions
EVENTS
Initialisation
    begin
        act : A ∧ AB
    end
Event EvtG1 ≙
    where
        grd : TRUE
    then
        act : AB
    end
Event EvtG2 ≙
    where
        grd : TRUE
    then
        act : B
    end
Event EvtG ≙
refines EvtG
    where
        grd : AB ∧ B
    then
        act : B
    end
END
```

#### Or-Refinement

When we have an or-refinement, it means that the parent goal is satisfied when one or more of the sub-goals have been satisfied. The `EvtG` event of the parent machine is refined into a new event `EvtG'` taking as pre-condition a formula expressing that one or more of the two sub-goals have been satisfied. It does not seem to be a generic approach here and the knowledge and competence of the analyst will play an important role. For instance in the example case described by Matoussi *et al.* in [Gervais et al., 2009], the guard of a refined `EvtG'` event, corresponding to a goal $G$ saying that some elements have to be localised in one way or another, uses the union of two sets, one for each of the sub-goals, saying that an element may be localised via GPS or via WIFI, and compare it to the set of all the elements:

$$\ldots \wedge LocalisedElements = (LocalisedByGPSElements \cup LocalisedByWIFIElements) \wedge \ldots$$

### 5.1.2 Second Phase

In the second phase, functional and non-functional goals are treated the same way. The main idea here is to say that an operation can be executed while the associated goal has not been satisfied (considering the non-functional properties too), which is the same as while it's post-condition has not been verified. However, this is not sufficient to ensure that an "Achieve" goal has been reached. A new event called "closing" is added with a guard equal to the post-condition (without the non-functional properties) of the goal to reach. So for the initial machine corresponding to the root goal $G$ we will have an event `EvtOpG` that can be executed while $G$ has not been reached and an event `Closing` that can be executed when $G$ is satisfied. This `Closing` event will finalize the system. As in the first phase, the machine will be refined following the refinement pattern used in the goal model and each level in the goal model will correspond to a machine in the Event-B model.

Note that in their example, Matoussi *et al.* in [Gervais et al., 2009] are working with sets and express the negation of the initial goal post-condition with universal quantifiers. The initial machine for an "Achieve" goal $G$ with a formal definition $A \Rightarrow \Diamond B$ will be:

Listing 5.3: Operationalization Event-B: initial machine

**MACHINE** EventBOperationalSpecification_level_0
**SEES** ModelContext
**VARIABLES**
**Manipulated data**
**INVARIANTS**
inv : Data types definitions
**EVENTS**

**Initialisation**
  **begin**
    act : $A$
  **end**
**Event** *EvtOpG* $\widehat{=}$
  **where**
    grd : $\neg B$
  **then**
    act : Do something that makes things going further
  **end**
**Event** *Closing* $\widehat{=}$
  **where**
    grd : B without non-functional properties
  **then**
    act : $Exit := OK$
  **end**
**END**

As in the first phase, the initial model will be refined according to the refinement patterns used in the goal model. The `Closing` event is refined as it without modification, and the sub-goals will be translated into events.

### Milestone-Driven Refinement

When a parent goal $G$ is refined into sub-goals $G_1, ..., G_n$ according to the milestone-driven refinement pattern, it means that the goal $G$ can be decomposed into $n$ steps and that $G$ is satisfied if the final step $G_n$ is reached. The sub-machine refining the initial machine defined for the second phase, like the one described in listing 5.3, will thus have `EvtOpG1,...,EvtOpGn` declared events where the pre-condition is the negation of the post-condition of the corresponding `EvtGi` event declared in the Event-B model coming from phase one (listing 5.2 in our example). The action will be "something that makes things going further" to the step $G_{i+1}$. Again, the approach does not seem to be generic, the action that "makes things going further" will depend of the goal $G_i$ and its definition will rely on the analyst's skills. The realization of the last sub-goal $G_n$ implies the realization of the patent goal $G$, so the last event `EvtOpGn` will refine the `EvtOpG` event of the parent machine. The refinement of goal $G$ following the pattern described in figure 5.1 will give a machine:

Listing 5.4: Operationalization Event-B: initial machine

**MACHINE** EventBOperationalSpecification_level_1
**REFINES** EventBOperationalSpecification_level_0
**SEES** ModelContext
**VARIABLES**
  **Manipulated data**
**INVARIANTS**
  inv : Data types definitions
**EVENTS**
**Initialisation**

**begin**
act : $A$
**end**
**Event** *EvtOpG1* $\widehat{=}$
**where**
grd : $\neg AB$
**then**
act : Do something that makes things going further
**end**
**Event** *EvtOpG2* $\widehat{=}$
**refines** *EvtOpG*
**where**
grd : $\neg B$
**then**
act : Do something that makes things going further
**end**
**Event** *Closing* $\widehat{=}$
**refines** *Closing*
**where**
grd : B without non-functional properties
**then**
act : $Exit := OK$
**end**
**END**

#### Or-Refinement

As for phase one, when we have an or-refinement, it means that the parent goal is satisfied when one or more of the sub-goals have been satisfied. The `EvtOpG` event of the parent machine is refined into a new event `EvtOpG`' taking as pre-condition the negation of the corresponding event in the Event-B model of phase one, possibly simplified and where possible ambiguities have been removed.

The two sub-goals are handled as in the general case by having a pre-condition equals to the negation of the post condition of the corresponding event in the model coming from phase one.

## 5.2 From Goal-Oriented Requirements to Event-B Specification: B. Aziz *et al.*

To derive an Event-B model from a KAOS model, Aziz *et al.* propose to include in Event-B the notion of triggered event. This new notion will be used to translate the next ($\circ$) and bounded sooner-or-later ($\Diamond_{\leqslant d}$) time operators used in the formal definition of requirements and expectations in KAOS, into Event-B events.

The first subsection introduces the notion of triggered event described in [Bicarregui et al., 2008]. The second subsection will then describe the operationalisation patterns that can be used to derive Event-B from KAOS requirements and expectations explained in [Aziz et al., 2009].

### 5.2.1 Notion of Triggered Event

In Event-B, the guards don't express obligations to execute the events. When more than one guard is true at the same time, the next executed event is chosen in a non-deterministic way. It means that scheduling of events is non-deterministic.

A limited notion of obligation is described by triggers. They are the dual of guards in the sense that when a guard is false, the event may not occur while when a trigger is true, the event must occur. In other terms, a trigger condition for an event is equivalent to a constraint on when other events are permitted. This constraint can be expressed by adding the negation of the trigger condition to the guards of all other events. Event-B trigger users need to pay attention, indeed trigger permit to impose an order of execution. But, by doing this they also add implicit constraints on the other events.

There a tree kinds of triggered-events, the *next-trigger*, the *eventually-trigger* and the *within-trigger*. The next-trigger is used for an event that will be the next performed event as soon as its trigger-condition becomes true. A within-triggered event will have to be executed within the $n$ next steps, whereas a step corresponds to the execution of an event. The eventually-trigger is modelled by within-trigger with a unbounded non-deterministic choice of $n$.

#### NEXT-Trigger

A next-trigger event $e$ with a trigger condition $T$ and an action $R$ is equivalent to a within-trigger with a $n$ equals to 0. It is notated :

$$\textbf{EVENT } e \textbf{ WHEN } T \textbf{ NEXT } R \textbf{ END}$$

In comparison with classical Event-B events, using next-trigger events may imposes to add restrictions on the other guards and trigger conditions to avoid deadlocks. We explain hereafter the interpretation of two classical events $e_1$ and $e_2$ used with two trigger-events $f_1$ and $f_2$ in classical Event-B. The four events are defined as :

$$\textbf{EVENT } e_1 \textbf{ WHEN } G_1 \textbf{ THEN } S_1 \textbf{ END}$$
$$\textbf{EVENT } e_2 \textbf{ WHEN } G_2 \textbf{ THEN } S_2 \textbf{ END}$$
$$\textbf{EVENT } f_1 \textbf{ WHEN } T_1 \textbf{ NEXT } R_1 \textbf{ END}$$
$$\textbf{EVENT } f_2 \textbf{ WHEN } T_2 \textbf{ NEXT } R_2 \textbf{ END}$$

The negation of the trigger conditions $T_1$ and $T_2$ will have to be added to the guards $G_1$ and $G_2$ to prevent the other events form being executed when a trigger has to occurs. But this is not enough, the negation of each trigger condition will also be added to the other trigger conditions to prevent the other trigger-events from being executed when one trigger condition is true. The equivalent of the four events here above in classical Event-B is :

**EVENT** $e_1$ **WHEN** $G_1 \wedge \neg T_1 \wedge \neg T_2$ **THEN** $S_1$ **END**
**EVENT** $e_2$ **WHEN** $G_2 \wedge \neg T_1 \wedge \neg T_2$ **THEN** $S_2$ **END**
**EVENT** $f_1$ **WHEN** $T_1 \wedge \neg T_2$ **THEN** $R_1$ **END**
**EVENT** $f_2$ **WHEN** $T_2 \wedge \neg T_1$ **THEN** $R_2$ **END**

A Event-B model composed only with trigger-events and classical-events will be said deadlock-free if and only if for every next-event $e_i$ with trigger condition $T_i$, the proposition $\neg(T_i \wedge T_j)$ is true for every $j \neq i$. For the example here above, the system is deadlock free if $\neg(T_1 \wedge T_2)$ is true. As next-trigger can be seen as a within-trigger with $n$ equals to 0, a more general definition of deadlock-free model is given for within-triggers in [Bicarregui et al., 2008].

### EVENTUALLY-Trigger

As said before, an eventually-trigger event is equivalent to an within-trigger event with a unbounded non-deterministic choice of $n$. In practise, the choice of $n$ is made when the trigger-condition becomes true and so the deadline will be set at that time and is only known internally. The notation for eventually-trigger events is :

**EVENT** $e$ **WHEN** $T$ **EVENTUALLY** $S$ **END**

### WITHIN-Trigger

Within-trigger are used for events that have to occur at most $n$ steps after a certain condition becomes true, as far as this condition is still true during the steps before the event effectively occurs. If the trigger condition becomes false during the $n$ steps before the event has occurs, the obligation is cancelled. A within-trigger event is notate as :

**EVENT** $e$ **WHEN** $T$ **WITHIN** $n$ **NEXT** $S$ **END**

### Triggered-Events with Guards

As for classical events, the triggered events may have a guard. As the trigger-condition will express the states where the event will have to occur, the guard-condition will express the states where the event may occur. The most general triggered event with trigger-condition $T$ and guard-condition $G$ can be noted:

**EVENT** $e$ **WHEN** $(T,G)$ **WITHIN** $n$ **NEXT** $S$ **END**

To be well formed, when the triggered-event is obliged, then it must be permitted, or in other words $T \Rightarrow G$. The classical events will then correspond to trigger-events with a false condition.

### Refinement of Triggered-Events

With classical-events, refining an abstract event $a$ in a more concrete event $b$, noted $a \sqsubseteq b$, means that proof obligations described in section 3.2.3 are respected. In particular, the guard strengthening proof obligation states that $b$'s guard is stronger than $a$'s guard, to ensure that when $b$ happens, so do $a$.

With triggered-events, there are two things that can be refined: the *duration* $n$ and the *trigger condition* $T$.

**Refinement of duration** Refining an event of a system means that the number of the possible states of the system will decrease, making the behaviour of the system more precise. In the case of triggered-events, it means that the maximal number of steps between the moment when the trigger condition becomes true and the moment when the event is effectively executed will decrease during the refinement process. Formally, if we have $T$ a trigger condition, $S$ a substitution and $m$ and $n$ two integers such as $0 \leqslant n \leqslant m$, then:

$$\begin{gathered}\textbf{EVENT } e \textbf{ WHEN } T \textbf{ EVENTUALLY } S \textbf{ END} \\ \sqsubseteq \textbf{EVENT } e1 \textbf{ WHEN } T \textbf{ WITHIN } m \textbf{ NEXT } S \textbf{ END} \\ \sqsubseteq \textbf{EVENT } e2 \textbf{ WHEN } T \textbf{ WITHIN } n \textbf{ NEXT } S \textbf{ END} \\ \sqsubseteq \textbf{EVENT } e3 \textbf{ WHEN } T \textbf{ NEXT } S \textbf{ END}\end{gathered}$$

**Refinement of the trigger predicate** If guard-conditions are strengthened during a refinement process, trigger-conditions, which are dual, will be weakened. This can be explained by the fact that adding a triggered-event to a model has as effect to add the negation of the trigger-condition to all other events of the model. Weakening a trigger-condition means then that all other guards will be enforced.

For an abstract trigger-event $e_a$ with a trigger-condition $T_a$ and a concrete event $e_b$ with a trigger-condition $T_b$ in a deadlock-free model, if $T_a \Rightarrow T_b$, than we have that:

$$\begin{gathered}\textbf{EVENT } e_a \textbf{ WHEN } T_a \textbf{ WITHIN } n \textbf{ NEXT } S \textbf{ END} \\ \sqsubseteq \textbf{EVENT } e_b \textbf{ WHEN } T_b \textbf{ WITHIN } n \textbf{ NEXT } S \textbf{ END}\end{gathered}$$

### Deadlock Freeness

In classical Event-B, a model is said deadlocked if it reaches a certain state where no guard is true, meaning that no event can be executed. With triggered-events, another kind of deadlock is possible if two events must occur at the same time. To avoid this, Bicarregui *et al.* [Bicarregui et al., 2008] introduce the notion of scheduling, which is briefly explained here.

Every event can be expressed as a WITHIN-trigger with a certain $n$. The idea is to associate an active counter, equals to $n$, to an event when its trigger-condition becomes true. This active counter is decreased each time an event occurs in the model. An event will be *schedulable* if, when a trigger-condition become true, there is enough space in the "execution queue" so that the event may occur, in other words, if there are at most $n$ other active counters with a value less than or equal to $n$. In the same way, a model will be schedulable if all its events are schedulable at all time.

### 5.2.2 Operationalisation Patterns

Aziz *et al.* [Aziz et al., 2009] reuse the notion of trigger-events to translate the next ($\circ$) and bounded sooner-or-later ($\Diamond_{\leqslant d}$) time operators used in the formal definition of requirements and expectations in KAOS, into Event-B events.

Table 5.1 presents the operationalisation patterns for the three most used goals types. $A$ and $B$ in the KAOS requirement's formal definition represents first-order logical formula defined over objects of the KAOS model. Those objects are translated into variables in the Event-B model and thus $A'$ represent the formula equivalent to $A$ defined over those variables and $B'$ represent the generalised substitution derived from predicate $B$, which will be seen as the post-condition of the substitution.

Table 5.1: Patterns for Operationalising Requirements into Event-B [Aziz et al., 2009]

| **Requirements** | **Formal Definition** | **Event-B Operationalisation** |
|---|---|---|
| Immediate Achieve | $A \Rightarrow \circ B$ | EVENT $e$ WHEN $A'$ NEXT $B'$ END |
| Bounded Achieve | $A \Rightarrow \Diamond_{\leqslant d} B$ | EVENT $e$ WHEN $A'$ WITHIN $d$ NEXT $B'$ END |
| Unbounded Achieve | $A \Rightarrow \Diamond B$ | EVENT $e$ WHEN $A'$ EVENTUALLY $B'$ END |

## 5.3 Deriving Event-based Security Policy from Declarative Security Requirements: R. De Landtsheer

De Landtsheer proposes in [Landtsheer, 2007a, Landtsheer and Ponsard, 2010] to translate linear temporal logic formula expressed exclusively with past operators into an event-based security policy expressed in Polpa. A syntax change can translate this Polpa policy to an Event-B model.

First we will present the linear temporal logic formulas admitted by this method. The second subsection briefly presents the Polpa element used by the method, more details about Polpa can be found in [Aziz et al., 2008]. The third describes the derivation procedure itself and the last subsection describes the syntactic changes to switch from Polpa to Event-B.

### 5.3.1 Linear Temporal Logic Formula

Temporal logic has already been explained in section 2.3.4. Due to the difficulty to manage the future (foreseen actions, schedule appropriate reaction, non-computability of infinite size models, *etc.* ), the formulas here are restricted to the past (see subsection 5.2 in [Landtsheer, 2007a] for more details). They may contain:

- Events represented as a predicated over typed variables.
- Logical connectors and $\wedge$, or $\vee$, not $\neg$ and implies $\rightarrow$.
- Quantifiers $\forall$ and $\exists$ used to specify the type of used variables.
- Temporal operators of the past since $\boldsymbol{S}$, has always been $\blacksquare$ and sometime in the past $\blacklozenge$.

For instance, a requirement for a file access control system could be expressed as follows:

$$(\forall u : Users)(\forall f : File)$$
$$open(u, f) \Rightarrow \neg forbidden(u, f) \ \boldsymbol{S} \ authorized(u, f)$$

Saying that a file can be opened by a user if he has received an authorisation for that file and if that authorisation has not been revoqued. As explained in section 2.3.4, $P \Rightarrow Q$ is used as a shorthand for $\Box(P \rightarrow Q)$.

### 5.3.2 Polpa

Polpa does with a given policy and a queue of events what regular expressions do with a characters pattern and a given string of characters. It will read a sequence of events and will accept or reject the events according to a given policy. We will present here the most important notions used in the rest of this section. More details about Polpa can be found in [Aziz et al., 2008]. To describe policies, Polpa uses three atomic constructions separated by the sequencing operator $(\cdot)$:

- Events are noted as in temporal logic, *e.g.* $open(u_0, f_0)$ represents an event in Polpa.

- Conditions which are non temporal assertions are placed between brackets and checked at runtime. If a condition is false, the events placed afterwards won't happen. For instance $[NOT\_FORB\_SINCE\_AUTH(u_0, f_0)] \cdot open(u_0, f_0)$ means that to open a file, some condition represented by $NOT\_FORB\_SINCE\_AUTH(u_0, f_0)$ has to hold.

- Actions are executable instructions used to update the internal state of the policy. They are represented between curly brackets, *e.g.* $authorized(u_0, f_0) \cdot \{NOT\_FORB\_SINCE\_AUTH(u_0, f_0) := true\}$.

Fragments of the policy may also be enclosed into loops, denoted by the $it(<\text{piece}>)$ keyword and disjunctive compositions, denoted by $par(<\text{piece}>, <\text{piece}>, ...)$, expressing the fact that all the pieces of the decomposition are acceptable. For instance, the Polpa policy generated for the requirement expressed before using this method will be:
$\{(\forall u : Users)(\forall f : File) NOT\_FORB\_SINCE\_AUTH(u, f) := false\} \cdot$
$it(par([NOT\_FORB\_SINCE\_AUTH(u_0, f_0)] \cdot open(u_0, f_0),$
$authorized(u_0, f_0) \cdot \{NOT\_FORB\_SINCE\_AUTH(u_0, f_0) := true\},$
$forbidden(u_0, f_0) \cdot \{NOT\_FORB\_SINCE\_AUTH(u_0, f_0) := false\}))$

### 5.3.3 Derivation Procedure

The main problem in this transformation is the switch from linear temporal logic formulas, which are expressed in a state-based logic where each predicate has a value at each time unit, to Polpa, which is an event-based logic where histories are described by a succession of events. It means that in event-based logic, there is a mapping between events and time units stating that each event corresponds to one time unit. It also means that time progresses only when events happen. To overcome this, at some point in the transformation, two assumptions are incorporated into the model. The first assumption, also called single input assumption, states that only one event can happen at a time. The second assumption states that each event is mapped to a time unit, in other words there is one event between two states in the model.

Figure 5.3 presents a data flow diagram of the transformation process. This transformation can be decomposed into three main steps:

- First, each temporal operator is removed from the initial formula and replaced by a corresponding state predicate which captures the value of this temporal operator throughout time.

- Secondly, for each state predicate, a start clause and a step clause are defined. The start clause defines the value of the state predicate at the beginning of the history. The step clause defines the value of the state predicate at each time point according to the previous values and the events occurring at this time.

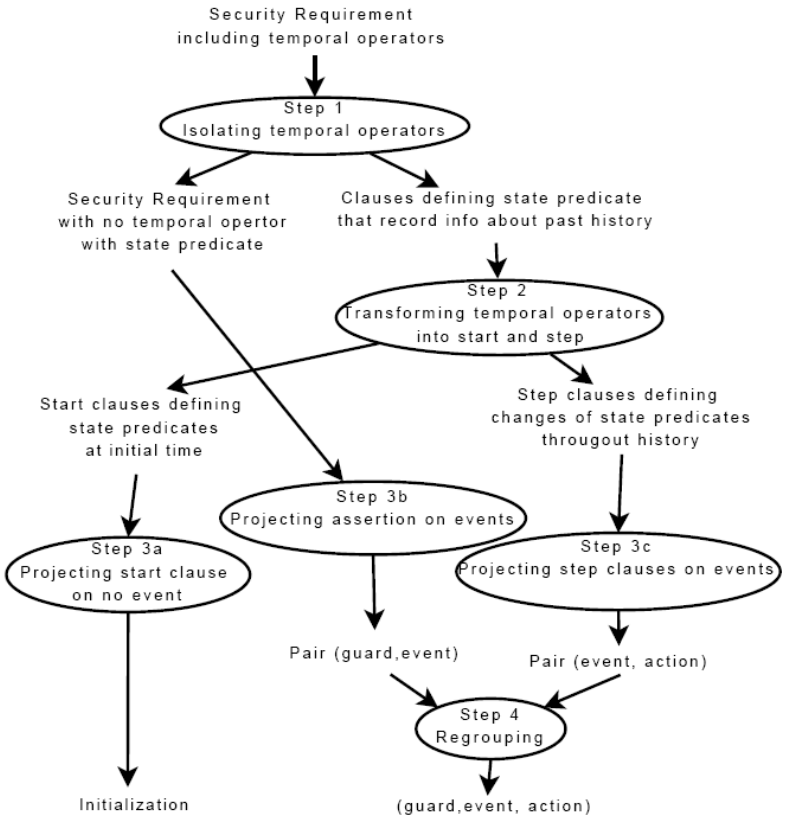


Figure 5.3: Data flow diagram of the translation process [Landtsheer and Ponsard, 2010]

- Finally, the assertions and clauses are projected onto the set of events. The principle of the projection is to take an event and consider that only this event happens. This is where the switch from state-based logic to event-based logic is done by using the single input assumption. Assertions are then simplified according to this assumption.

### Isolating Temporal Operators

In this step, the initial assertion is simplified by replacing the temporal operators by a corresponding state predicate. This state predicate must be constructed such that the semantics of the temporal operators is respected. As seen in table 2.2 on page 19, the past operators work with values of the predicates before a current time position $i$, but also with the value of those predicates at the time position $i$. For instance, $\blacklozenge P$ is true at a time position $i$ if $P$ is true at all times before $i$ and if $P$ is true at the current time position $i$ too. In this step, $\blacklozenge P$ will thus be replaced by $HISTORICALLY\_P \wedge P$.

Table 5.2 gives the translation rules for the considered past time operators.

Table 5.2: Step1: translation rules

| **Temporal operator** | **Substitute** |
|---|---|
| $\blacklozenge P$ | $HISTORICALLY\_P \wedge P$ |
| $P \ \boldsymbol{S} \ Q$ | $(P\_SINCE\_Q \wedge P) \vee Q$ |
| $\blacksquare P$ | $ONCE\_P \vee P$ |

In the file access control system requirement, the since operator ($P \boldsymbol{S} Q$) is the only past time operator. The considered sub-formula is:

$\neg forbidden(u, f) \ \boldsymbol{S} \ authorized(u, f)$

It will be replaced by:

$(NOT\_FORB\_SINCE\_AUTH(u, f) \wedge \neg forbidden(u, f))$
$\vee authorized(u, f)$

The final assertion will be:

$(\forall u : Users)(\forall f : File)$
$open(u, f) \Rightarrow ((NOT\_FORB\_SINCE\_AUTH(u, f) \wedge \neg forbidden(u, f))$
$\vee authorized(u, f))$

### Defining Start and Step Clauses

In this step, a start clause and a step clause are defined for each state predicate generated in the first step. The start clause defines the value of the state predicate at the beginning of the history. The step clause defines the value of the state predicate at each time point according to the previous values and the events occurring at this time. The initial assertion is not modified in this step.

Table 5.3: Step2: generation rules

| **Temporal operator** | **Start clause** | **Step clause** |
|---|---|---|
| $\blacklozenge P$ | $P$ | $HISTORICALLY\_P \wedge P$ |
| $P \ \boldsymbol{S} \ Q$ | $Q$ | $(P\_SINCE\_Q \wedge P) \vee Q$ |
| $\blacksquare P$ | $P$ | $ONCE\_P \vee P$ |

Table 5.3 gives the generation rules for the considered past time operators. For instance, the state predicate $NOT\_FORB\_SINCE\_AUTH(u, f)$ will have the start clause:
$authorized(u, f)$

And the step clause:
$(NOT\_FORB\_SINCE\_AUTH(u, f) \wedge \neg forbidden(u, f))$
$\vee authorized(u, f)$

### Projecting State-Based Models onto Event-Based Logics

We have now different elements: the simplified initial assertion, a set of start clauses and a set of step clauses. The switch from state-based to event-based logic in the model is done at this step, by considering that time progresses only when events are true and that there is only one true event at the same time. Each event will be considered in turn and the model will be projected on this event by considering that all others events are false. In the example, there are three events: *open*, *authorized* and $forbidden$. The projected assertion will be simplified and all projections will be regrouped afterwards. This step may be decomposed in three sub-steps: projecting the start clauses, projecting the assertion and projecting the step clauses.

**Projecting start clauses** By convention, no event occurs at the initial time. Start clauses are simplified by replacing each occurrence of event by false. All start clauses have the pattern $\forall * STATE\_PREDICATE(*) := P(*)$ to initialize the value of the state predicates. The simplification will take place in the right part of the assignment operator and will give for the example introduced before:

$$\begin{array}{l} \forall u : User \\ \quad \forall f : File \\ \qquad NOT\_FORB\_SINCE\_AUTH(u, f) := false \end{array}$$

**Projecting assertion on events** Each event will be considered in turn, fixing arbitrary parameters and projecting the assertion on it using the $project(P(x))$ function. Projecting the assertion on an event means that all events are false, except the one considered with its fixed parameters. To complete this projection, different instantiations to the universally quantified variables are considered. The original quantifications in the projected assertion are replaced according to the rules:

$$\begin{array}{rcl} project((\forall x : X)P(x)) & \rightsquigarrow & (\forall x_c : X)project(P(x_c)) \wedge \wedge_e project(P(x_e)) \\ project((\exists x : X)P(x)) & \rightsquigarrow & (\exists x_c : X)project(P(x_c)) \vee \vee_e project(P(x_e)) \end{array}$$

Where $x_c$ is a quantified variable denoting "all the other values than the ones afterwards" and $x_e$ correspond to the constants from the event with type $X$. If there is no more quantifier in $P(x)$, $project(P(x))$ will simply return $P(x)$. For the outer quantifiers, a conjunct or disjunct is added in the final conjunct (after $\wedge_e$) or disjunct (after $\vee_e$) if at least one event is true in it. The reason is that, if no event is true in a sub-formula, then the sub-formula is constant, so that its value has not changed since the previous time unit.

All events will then be simplified according to the principle of projection saying that only the considered event with the fixed parameters is true.

In the example, the *open* event will have arbitrary parameters $u_0$ and $f_0$. The assertion in subsection 5.3.3 will be projected on this event, giving:

$$
\begin{aligned}
(\forall u_q : User) & \left( \begin{array}{l} (\forall f_q : File) \\ \left( \begin{array}{l} open(u_q, f_q) \\ \quad \Rightarrow (NOT\_FORB\_SINCE\_AUTH(u_q, f_q) \\ \qquad \wedge \neg forbidden(u_q, f_q)) \\ \quad \vee authorized(u_q, f_q) \end{array} \right) \\ \wedge \left( \begin{array}{l} open(u_q, f_0) \\ \quad \Rightarrow (NOT\_FORB\_SINCE\_AUTH(u_q, f_0) \\ \qquad \wedge \neg forbidden(u_q, f_0)) \\ \quad \vee authorized(u_q, f_0) \end{array} \right) \end{array} \right) \\
\wedge (\forall f_q : File) & \left( \begin{array}{l} open(u_0, f_q) \\ \quad \Rightarrow (NOT\_FORB\_SINCE\_AUTH(u_0, f_q) \\ \qquad \wedge \neg forbidden(u_0, f_q)) \\ \quad \vee authorized(u_0, f_q) \end{array} \right) \\
\wedge & \left( \begin{array}{l} open(u_0, f_0) \\ \quad \Rightarrow (NOT\_FORB\_SINCE\_AUTH(u_0, f_0) \\ \qquad \wedge \neg forbidden(u_0, f_0)) \\ \quad \vee authorized(u_0, f_0) \end{array} \right)
\end{aligned}
$$

All events that are not $open(u_0, f_0)$ are $false$. After simplification, this formula became:

$$
\begin{aligned}
(\forall u_q : User) & \left( \begin{array}{l} (\forall f_q : File) \\ \big( false \Rightarrow NOT\_FORB\_SINCE\_AUTH(u_q, f_q) \big) \\ \wedge \big( false \Rightarrow NOT\_FORB\_SINCE\_AUTH(u_q, f_0) \big) \end{array} \right) \\
\wedge (\forall f_q : File) & \big( false \Rightarrow NOT\_FORB\_SINCE\_AUTH(u_0, f_q) \big) \\
\wedge & \big( true \Rightarrow NOT\_FORB\_SINCE\_AUTH(u_0, f_0) \big)
\end{aligned}
$$

The final result of the projection is then:

$$NOT\_FORB\_SINCE\_AUTH(u_0, f_0)$$

Similar process is done on the *allowed* and $forbidden$ events, giving a simplified condition *true*, meaning that those events can always be accepted by the policy.

**Projecting step clauses on events** Step clauses include an assignment operator, which can be simplified if the left and right members of the assignment are equal. In that case, the assignment is replaced by a constant *true*. The projection of the step clause defined in subsection 5.3.3, on the event $authorized(u_0, f_0)$ give:

$$authorized(u_0, f_0) \cdot \{NOT\_FORB\_SINCE\_AUTH(u_0, f_0) := (NOT\_FORB\_SINCE\_AUTH(u_0, f_0) \wedge \neg false) \vee true\}$$

Which can be simplified by:

$$authorized(u_0, f_0) \cdot \{NOT\_FORB\_SINCE\_AUTH(u_0, f_0) := true\}$$

The projection on the event $forbidden(u_0, f_0)$ give:

$$forbidden(u_0, f_0) \cdot \{NOT\_FORB\_SINCE\_AUTH(u_0, f_0) := false\}$$

**Regrouping**

The last step regroups start clauses into an initialization part and regroups projection of conditions and step clauses by events. A Polpa policy is build from those projections. The policy generated for the file access control system is :
$\{(\forall u : Users)(\forall f : File)NOT\_FORB\_SINCE\_AUTH(u, f) := false\} \cdot$
$it(par([NOT\_FORB\_SINCE\_AUTH(u_0, f_0)] \cdot open(u_0, f_0),$
$authorized(u_0, f_0) \cdot \{NOT\_FORB\_SINCE\_AUTH(u_0, f_0) := true\},$
$forbidden(u_0, f_0) \cdot \{NOT\_FORB\_SINCE\_AUTH(u_0, f_0) := false\}))$
Two more complete examples of derivation can be found in [Landtsheer, 2007a].

### 5.3.4 Syntactic Changes from Polpa to Event-B

As in Event-B, there are events, actions and guards in the policies generated by this derivation procedure. The translation from Polpa to Event-B may be then a simple syntactic change presented in table 5.4. The state predicates generated by the derivation procedure will become internal variables of the machine.

Table 5.4: Syntactic changes from Polpa to Event-B [Landtsheer, 2007a]

| **Polpa element** | **Event-B element** |
|---|---|
| Initialization code | `INITIALISATION` event |
| Condition · Event · Action | Event-B event |
| Condition in Polpa clause | `WHERE` clause of the Event-B event |
| Event in Polpa clause | Name and parameters (`ANY`) |
| Action in Polpa clause | `THEN` clause of the Event-B event |

The Event-B machine derived from the generated Polpa policy for the file access control system is given in listing 5.5. The seen context, which is not detailed here, declare the `file` and `user` sets.

Listing 5.5: Event-B machine derived from Polpa policy

**MACHINE** Simple_Authorization_Scheme
**SEES** Ctx0
**VARIABLES**
NOT_FORBIDDEN_SINCE_AUTHORIZED
**INVARIANTS**
inv1 : $NOT\_FORBIDDEN\_SINCE\_AUTHORIZED \subseteq ((file \times user) \rightarrow BOOL)$
**EVENTS**
**Initialisation**
**begin**
act1 : $(\forall qfile \cdot qfile \in file)(\forall quser \cdot quser \in user)$ $NOT\_FORBIDDEN\_SINCE\_AUTHORIZED(qfile, quser) := FALSE$
**end**
**Event** *open* $\widehat{=}$
**any**
$f$
$u$
**where**
grd1 : $f \in file$
grd2 : $u \in user$
grd3 : $NOT\_FORBIDDEN\_SINCE\_AUTHORIZED(f, u)$
**then**
skip
**end**
**Event** *forbidden* $\widehat{=}$
**any**
$f$
$u$
**where**
grd1 : $f \in file$
grd2 : $u \in user$
**then**
act1 : $NOT\_FORBIDDEN\_SINCE\_AUTHORIZED(f, u) := FALSE$
**end**
**Event** *authorized* $\widehat{=}$
**any**
$f$
$u$
**where**
grd1 : $f \in file$
grd2 : $u \in user$
**then**
act1 : $NOT\_FORBIDDEN\_SINCE\_AUTHORIZED(f, u) := TRUE$
**end**
**END**

## 5.4 Comparison of the Approaches with our Proposed Approach

This section compares the three methods presented in this chapter with our method described in chapter 4. We carry out our comparison based on a number of criteria and then give a global synthesis under a tabular form.

### 5.4.1 Traceability

Traceability is very interesting to bridge a requirement model and a specification model. To ensure that requirements have effectively been translated into specifications, but also, and particularly with formal Event-B specifications, to "explain" the specification by elements coming from the requirements. It is also very important to have traceability links between requirements and specifications when requirements are discovered or corrected during the specifications phase if incompletenesses or lacks are pointed out by the specifications.

- Matoussi's approach works in two steps. First giving an Event-B representation of a KAOS model to then operationalize, in the KAOS sense of the word, this model. All the events defined in the first step will correspond to a goal in the KAOS diagram. The traceability is direct here. In the second step, each event, except the closing event, will correspond to the operationalization of one goal. The traceability is direct too.

- In the method proposed by Aziz *et al.* each requirement/expectation expressed using one of the three Achieve goal type will be translated into one trigger-event. Even if traceability issue is not addressed in the method, the traceability here is direct between one requirement/-expectation and one trigger-event.

- The traceability issue is not either addressed in De Landtsheer's method. The traceability here is possible between one linear temporal logic formula expressed using past operators and a set of Event-B elements (containing variables, invariants, actions in the initialisation event and events).

- In our proposed approach, traceability between Event-B specifications and KAOS requirements is mandatory to keep the models consistent with each other. It is guaranteed by a set of traceability rules that has to be respected. In the prototype we developed , the traceability links are present in a redefined Event-B meta-model and the traceability rules will be, in future developments, represented as constraints in this meta-model.

### 5.4.2 Models Evolution

As explained in the previous sub-section, both requirements and specifications models may evolve, even if the Event-B specifications has already been derived from KAOS requirements, if incompletenesses or lacks are pointed out by the specifications or if the development is incremental for instance.

It is thus interesting to see if the different methods support evolution of one model and reflect this evolution on the other.

- The method proposed by Matoussi is executed once from the KAOS goal model to give an Event-B specification. Although the first phase may work incrementally, since a refinement level in the KAOS goal tree corresponds to a machine refinement in the Event-B model so the two models may be refined in parallel, this is not the case for the second phase. Modifications in one goal at one level will need to replay the first phase of the method from the goal's refinement level and the replay of all the second phase of the method. Going on the other way is also more difficult since no traceability mechanisms are defined. Reflecting a modification of the Event-B model in the KAOS model will be left for the analyst.

- The method proposed by Aziz *et al.* supports only one type of increment: adding a new requirement/expectation expressed using one of the three Achieve goal type will give a new trigger-event in the Event-B machine. If one requirement/expectation is modified in the KAOS model, a corresponding trigger-event will be regenerated and will override the old one in the Event-B model. In this case too, if the Event-B model is modified, the analyst will have to manually modify the KAOS model.

- As for Aziz's *et al.* method, De Landtsheer's method supports only one type of increment: a new requirement/expectation with a formal definition expressed using past time operators will give a new set of Event-B elements. Here too, modifying the formal definition of one requirement/expectation in the KAOS model will need a replay of the method to get a new set of Event-B elements that will override the previous set associated to the formal definition. If the Event-B model is modified, it will be very hard to reflect this modification in the KAOS model.

- Our proposed method supports incremental development of requirements and reflection of this development in the Event-B model thanks to the traceability mechanism (see section 4.7 on page 70). This avoid complete regeneration of the Event-B model each time an element of the KAOS model is modified. Altough this path has not been explored further, it is also theoretically possible to reflect in the KAOS model changes of the Event-B model. Traceability rules will ensure that the two models are consistent one with the other and traceability links will permit to the analyst to see in the KAOS model which elements are impacted by the changes of the Event-B model.

### 5.4.3 Scalability

As Event-B models are for now quite monolithic (see sub-section 3.2.4 on page 41), it is interesting here to see how the different approaches manage the complexity and the size of the automatically or semi-automatically generated the Event-B model.

- In Matoussi's method, everything coming from the KAOS model is put into two Event-B machines, one for each phase of the method. The approach is semi-automatic and the increasing size of the Event-B machine is managed through a refinement strategy. In the two phases of the method, an initial machine will be refined and complexity will be added to those machine at each refinement step.
- In the method proposed by Aziz *et al.* all the requirements/expectations expressed using one of the three Achieve goal type will be translated automatically in trigger-events. All those events will be put in the same machine. After applying this method, the Event-B model will thus contain one machine with all the trigger-events coming from the requirements/expectations.
- In De Landtsheer's method, all the requirements/expectations are automatically translated into sets of Event-B elements. All those elements are placed in the same machine that will be the only machine present in the Event-B model after application of the method.
- In our proposed method the transformation is, in general, semi-automatic. To manage the size of the different machines present in the Event-B model, we have chosen to apply an initial decomposition in place of having decompositions latter in the Event-B specification process. This initial decomposition has the advantage to be "motivated by requirements", meaning that the agent machines resulting of the decomposition are linked to the KAOS agents of the KAOS model. Agent machines are thus more than just machines in the Event-B sense of the term, they specify the behaviour of a KAOS agent defined in the requirements.

### 5.4.4 Restrictions

All the methods described in this chapter are limited to a subset of KAOS elements.

- Matoussi's approach is for now limited to KAOS models using only one goal type, the unbounded Achieve goals corresponding to the formal definition pattern $A \Rightarrow \Diamond B$, and construct exclusively with two refinement patterns, the milestone refinement and the or-refinement.

- The method proposed by Aziz *et al.* uses the notion of trigger-event, which is not standard in Event-B, to translate the three more used goal patterns, the immediate Achieve, the unbounded Achieve and the bounded Achieve into triggered events. Although more patterns can be discovered, the notion of trigger-event hide a complex mechanism of event scheduling that may quickly introduce misinterpretation errors. Indeed, introducing an event with a trigger condition means that restrictions are introduced on all other events.

- De Landtsheer's procedure was created to translate linear temporal logic into a Polpa policy using Polpa policy language, a language to express acceptable sequences of events. As underlined by De Landtsheer the notions of events, conditions and actions present in Polpa are similar to those present in Event-B with a syntactic translation. The limitation is that the procedure works exclusively with the since ($A \ \boldsymbol{S} \ B$), the always been ($\blacksquare A$) and the once ($\blacklozenge A$) temporal operators. But since this procedure takes only past linear temporal logic formula on input, it may only be used to translate a requirement's/expectation's formal definition into Event-B, without taking care of the rest of the KAOS model.

- In our proposed method, delegating the transformation of the requirements/expectations into Event-B specifications to the analyst allows to use unrestricted KAOS constructions. Of course, this means that step 3 is, for now, not automatic in our approach and that the analyst will have to manually derive Event-B from KAOS requirements/expectations. This may seem problematic since the misinterpretation error risk is reintroduced here, but in practice the requirement/expectations are detailed enough to avoid such errors. Moreover, in practice the formal linear temporal logic definition is not always established for the requirements/expectations in KAOS models. Usually, the goal type, which corresponds to a linear temporal logic formula pattern, and the name of the requirement/expectation is enough to understand what the goal means, *e.g. Achieve[Pump Started WHEN HighWater EXPT if Gas Detected]* will formally mean $(HighWater \wedge \neg GasDetected) \Rightarrow \Diamond PumpStarted$.

Even if our method is not fully automatic in its description, it is possible to use at step 3 every method that transforms requirement/expectation into Event-B. If we use for instance the methods proposed by De Landtsheer or Aziz *et al.* at step 3, our method becomes then fully automatic, but will have to be restricted to the KAOS elements permitted by those method. It is also possible to combine those methods, transforming past linear temporal logic formula with De Landtsheer's method and the three kinds of Achieve goals with Aziz's *et al.* method.

### 5.4.5 Summary

Table 5.5 shows a brief recap of the comparison of our approach presented in chapter 4 and the three approaches presented in this chapter. Some of the points have been split in the table to make it more readable: the model evolution sub-section covers the monolithic[1] and increment supports criteria of the table; the scalability sub-section covers the scalability and automatic criteria of the table. A more detailed discussion will be presented in the conclusion chapter of this master thesis.

[1] By monolithic, we mean that the method does not describe mechanisms to update one model if the other is modified

Table 5.5: Comparison of the different approaches

| **Criterion** | **Matoussi** | **Aziz *et al.*** | **De Landtsheer** | **Proposed approach** |
|---|---|---|---|---|
| **Traceability** | Possible for events | Possible for trigger-events | Possible for sets of Event-B elements | Mandatory to ensure consistency between models |
| **Monolithic** | Yes, re-generation needed | Yes, re-generation needed | Yes, re-generation needed | No, traceability mechanisms avoid complete re-generation |
| **Support increment** | Unidirectional, if in the first phase | Unidirectional, a new goal will give a new trigger-event | Unidirectional, a new formal definition will give a new Event-B elements set | Bidirectional, modifications in one model may be reflected in the other |
| **Scalability** | Two machines, constructed by refinement | One machine | One machine | Initial decomposition |
| **Automatic** | Semi-automatic | Automatic | Automatic | Semi-automatic |
| **Restrictions** | Restricted to *Immediate Achieve* goals and *milestone*/*or* refinement patterns | Restricted to three *Achieve* goal types | Goal must be formally defined with temporal past operators | No restriction on input, but will need the analyst's skills |

# Chapter 6

# Conclusion

In this master thesis, we have worked on the key problem of bridging the gap between requirements and formal specification with a strong model driven approach. We have explored this problem in the context of the KAOS goal oriented requirement engineering approach and Event-B, both methods being industrially used. After having defined what we wanted to achieve: a flexible method, automatic when possible, that support incremental and iterative development, with respect to the KAOS and Event-B semantics and a fine grained traceability. And reviewed existing methods against those criteria, we proposed our own approach, prototyped it and experimented with it.

**Our proposed method** is based on the idea of refining the KAOS agent's agent behaviour, produced as outcome of a goal-oriented analysis. As agent's behaviours are expressed in KAOS as a set of control and monitor links between the agents and the data declared in the object model. An initial machine and an initial context are created in Event-B to represent the KAOS object model in Event-B. Each KAOS agent will be associated with an agent machine, and the initial machine will be decomposed into those agent machines. A KAOS agent that controls or monitors a data will have in its agent machine the variable, invariant and update event derived from this data declaration. If the agent monitors it, the update will be marked as external, *i.e.* the agent will not be responsible of the concrete implementation of this event and will thus not refine it in its sub-machine, but it will be notified if the data change in time. If the agent controls the data, the update event will be marked as internal, *i.e.* the agent will be responsible of the concrete implementation of this event and will refine it in its sub-machine. Agent machines are then refined, and so are their internal events to make correspond the update of the data to the behaviours declared in the requirements/expectations placed under the responsibility of the KAOS agent. This last step is not automatic and will rely on the skills of the analyst that performs the switch from KAOS to Event-B.

The approach proposed in this master thesis is **semi-formal**, we rather

preferred here to focus on a **flexible way** to manage both KAOS requirements and Event-B specifications than to have a fully automated way to derive the second from the first. One of the reason of this choice was that even if KAOS includes a formal layer, permitting to define requirements and expectations using linear temporal logic, this formal definition is actually rarely done in practice. By linking the two models through **traceability links** and by constraining those links with a set of traceability rules, changing one model and keeping the other accurate is more easy. Although, updating the Event-B model when the requirements change is possible with the three techniques presented in chapter 5 (by regenerating the Event-B model most of the time), going on the other way is not possible. This has not been discussed further here, but it is left as an open path for future works. Such a need may be motivated in an incremental development for instance where requirements and specifications are elaborated step by step.

Another issue is **scalability**. As Event-B model may be very huge, even if there are techniques currently under research to decompose an Event-B model, they don't give a motivated reason or criteria to decompose a model in a systematic way. In contrast to the methods presented in chapter 5, where all is put in one or two machines, we propose with our approach to have a "requirement motivated" initial decomposition of the model. So each part, *i.e.* each agent machine, may be refined separately and in parallel.

**The semantic switch** between the KAOS open-world, where all is possible except what has been explicitly forbidden, to the Event-B closed-world, where only what has been declared is permitted, has not been discussed here. However, the idea of decomposing the Event-B initial machine into several agent machines, where each agent will give a software component running in parallel with the other components does not seem to contradict the KAOS semantic of a system, defined as the parallelization of the agent's behaviours. An open question is the signification of a re-composed Event-B machine, as described in subsection 4.5 on page 64, with regards to the KAOS semantic?

A first prototype has been elaborated with the purpose to prove feasibility and **automation** of the proposed approach. This prototype will need to be completed, as discussed in section 4.8 on page 74, in particular the verification of the traceability rules and the implementation of the link to the RODIN Event-B tool.

Finally, even if it could be already partially achieved by using the procedure described in section 5.3 on page 89, the method itself will need to be improved, by establishing a formal derivation of the KAOS requirements and expectations.

# Appendix A

# Mine pump example

This annex present the complete machines of the mine pump example described in chapter 4.

Listing A.1: Mine pump example: Initial context

**CONTEXT** MineContext
**SETS**
ONOFF
LEVEL
MINE_SET
**CONSTANTS**
ON
OFF
LOW
MEDIUM
HIGH
M
**AXIOMS**
axm1 : $partition(ONOFF, \{ON\}, \{OFF\})$
axm2 : $partition(LEVEL, \{LOW\}, \{MEDIUM\}, \{HIGH\})$
axm3 : $partition(MINE\_SET, \{M\})$
**END**

Listing A.2: Mine pump example: Initial machine

**MACHINE** MinePump
**SEES** MineContext
**VARIABLES**
MINE
pump
bell
methane
waterLevel
**INVARIANTS**
inv1 : $MINE \in \mathbb{P}(MINE\_SET)$
inv2 : $pump \in MINE \rightarrow ONOFF$
inv3 : $bell \in MINE \rightarrow BOOL$

inv4 : $methane \in MINE \rightarrow BOOL$
inv5 : $waterLevel \in MINE \rightarrow LEVEL$
inv6 : $dom(pump) = MINE$
inv7 : $dom(bell) = MINE$
inv8 : $dom(methane) = MINE$
inv9 : $dom(waterLevel) = MINE$
inv10 : $\forall m \cdot (m \in MINE) \Rightarrow (pump(m) = ON \lor pump(m) = OFF)$
inv11 : $MINE \subseteq MINE\_SET$
inv12 : $\forall m \cdot (m \in MINE) \Rightarrow$
$(waterLevel(m) = LOW \lor waterLevel(m) = MEDIUM \lor waterLevel(m) = HIGH)$

**EVENTS**
**Initialisation**
**begin**
act1 : $MINE := \varnothing$
act2 : $pump := \varnothing$
act3 : $bell := \varnothing$
act4 : $methane := \varnothing$
act5 : $waterLevel := \varnothing$
**end**
**Event** $updatePump \mathrel{\widehat{=}}$
**any**
$m$
**where**
grd1 : $m \in MINE$
grd2 : $pump(m) = ON \lor pump(m) = OFF$
**then**
act1 : $pump :\mid (pump(m) = OFF \land pump' = (pump \leftarrow \{m \mapsto ON\}))$
$\lor (pump(m) = ON \land pump' = (pump \leftarrow \{m \mapsto OFF\}))$
**end**
**Event** $updateBell \mathrel{\widehat{=}}$
**any**
$m$
$status$
**where**
grd1 : $m \in MINE$
grd2 : $status \in BOOL$
**then**
act1 : $bell(m) := status$
**end**
**Event** $updateMethane \mathrel{\widehat{=}}$
**any**
$m$
$status$
**where**
grd1 : $m \in MINE$
grd2 : $status \in BOOL$
**then**
act1 : $methane(m) := status$
**end**
**Event** $updateWaterLevel \mathrel{\widehat{=}}$
**any**
$m$
**where**
grd1 : $m \in MINE$
grd2 : $waterLevel(m) = LOW \lor waterLevel(m) = MEDIUM \lor waterLevel(m) = HIGH$
**then**

act1 : $waterLevel :|$
$(waterLevel(m) = HIGH \land waterLevel' = (waterLevel \lhd\!\!- \{m \mapsto MEDIUM\}))$
$\lor (waterLevel(m) = MEDIUM \land waterLevel' = (waterLevel \lhd\!\!- \{m \mapsto LOW\}))$
$\lor (waterLevel(m) = MEDIUM \land waterLevel' = (waterLevel \lhd\!\!- \{m \mapsto HIGH\}))$
$\lor (waterLevel(m) = LOW \land waterLevel' = (waterLevel \lhd\!\!- \{m \mapsto MEDIUM\}))$

**end**

**Event** $addMine \mathrel{\widehat{=}}$

**when**

grd1 : $MINE = \varnothing$

**then**

act1 : $MINE := \{M\}$
act2 : $pump(M) := OFF$
act3 : $bell(M) := FALSE$
act4 : $methane(M) := FALSE$
act5 : $waterLevel(M) := LOW$

**end**

**END**

Listing A.3: Mine pump example: PumpController machine

**MACHINE** PumpController
**SEES** MineContext
**VARIABLES**

`methane` Shared variable, DO NOT REFINE
`waterLevel` Shared variable, DO NOT REFINE
`pump` Shared variable, DO NOT REFINE
`MINE` Shared variable, DO NOT REFINE

**INVARIANTS**

typing_methane : $methane \in \mathbb{P}(MINE\_SET \times BOOL)$
typing_waterLevel : $waterLevel \in \mathbb{P}(MINE\_SET \times LEVEL)$
typing_pump : $pump \in \mathbb{P}(MINE\_SET \times ONOFF)$
typing_MINE : $MINE \in \mathbb{P}(MINE\_SET)$
MinePump_inv1 : $MINE \in \mathbb{P}(MINE\_SET)$
MinePump_inv2 : $pump \in MINE \to ONOFF$
MinePump_inv4 : $methane \in MINE \to BOOL$
MinePump_inv5 : $waterLevel \in MINE \to LEVEL$
MinePump_inv6 : $dom(pump) = MINE$
MinePump_inv8 : $dom(methane) = MINE$
MinePump_inv9 : $dom(waterLevel) = MINE$

**EVENTS**
**Initialisation**

**begin**

act1 : $MINE := \varnothing$
act2 : $pump := \varnothing$
act4 : $methane := \varnothing$
act5 : $waterLevel := \varnothing$

**end**

**Event** $updatePump \mathrel{\widehat{=}}$

**any**
$m$
**where**
grd1 : $m \in MINE$
**then**
act1 : $pump :|$
$((methane(m) = FALSE \wedge waterLevel(m) = HIGH) \Rightarrow pump'(m) = ON)$
$\wedge (methane(m) = TRUE \Rightarrow pump'(m) = OFF)$
$\wedge ((methane(m) = FALSE \wedge waterLevel(m) = LOW) \Rightarrow pump'(m) = OFF)$
**end**

**Event** *updateMethane* $\widehat{=}$
External event, DO NOT REFINE
**any**
$m$
$status$
**where**
grd1 : $m \in MINE$
grd2 : $status \in BOOL$
**then**
act1 : $methane(m) := status$
**end**

**Event** *updateWaterLevel* $\widehat{=}$
External event, DO NOT REFINE
**any**
$m$
**where**
grd1 : $m \in MINE$
**then**
act1 : $waterLevel :|$
$(waterLevel(m) = HIGH \Rightarrow waterLevel'(m) = MEDIUM)$
$\wedge (waterLevel(m) = MEDIUM \Rightarrow (waterLevel'(m) = LOW \vee waterLevel'(m) = HIGH))$
$\wedge (waterLevel(m) = LOW \Rightarrow waterLevel'(m) = MEDIUM)$
**end**

**Event** *addMine* $\widehat{=}$
External event, DO NOT REFINE
**when**
grd1 : $MINE = \varnothing$
**then**
act1 : $MINE := \{M\}$
act2 : $pump(M) := OFF$
act4 : $methane(M) := FALSE$
act5 : $waterLevel(M) := LOW$
**end**

**END**

Listing A.4: Mine pump example: WaterLevelSensor machine

**MACHINE** WaterLevelSensor
**SEES** MineContext
**VARIABLES**
`waterLevel` Shared variable, DO NOT REFINE
`MINE` Shared variable, DO NOT REFINE
**INVARIANTS**
`typing_waterLevel` : $waterLevel \in \mathbb{P}(MINE\_SET \times LEVEL)$

typing_MINE : $MINE \in \mathbb{P}(MINE\_SET)$
MinePump_inv1 : $MINE \in \mathbb{P}(MINE\_SET)$
MinePump_inv5 : $waterLevel \in MINE \rightarrow LEVEL$
MinePump_inv9 : $dom(waterLevel) = MINE$

**EVENTS**
**Initialisation**
**begin**
act1 : $MINE := \varnothing$
act5 : $waterLevel := \varnothing$
**end**
**Event** *updateWaterLevel* $\widehat{=}$
**any**
$m$
**where**
grd1 : $m \in MINE$
**then**
act1 : $waterLevel :|$
$(waterLevel(m) = HIGH \Rightarrow waterLevel'(m) = MEDIUM)$
$\wedge (waterLevel(m) = MEDIUM \Rightarrow (waterLevel'(m) = LOW \vee waterLevel'(m) = HIGH))$
$\wedge (waterLevel(m) = LOW \Rightarrow waterLevel'(m) = MEDIUM)$
**end**
**Event** *addMine* $\widehat{=}$
External event, DO NOT REFINE
**when**
grd1 : $MINE = \varnothing$
**then**
act1 : $MINE := \{M\}$
act5 : $waterLevel(M) := LOW$
**end**
**END**

Listing A.5: Mine pump example: AlarmController machine

**MACHINE** AlarmController
**SEES** MineContext
**VARIABLES**
`bell` Shared variable, DO NOT REFINE
`methane` Shared variable, DO NOT REFINE
`MINE` Shared variable, DO NOT REFINE
**INVARIANTS**
typing_bell : $bell \in \mathbb{P}(MINE\_SET \times BOOL)$
typing_methane : $methane \in \mathbb{P}(MINE\_SET \times BOOL)$
typing_MINE : $MINE \in \mathbb{P}(MINE\_SET)$
MinePump_inv1 : $MINE \in \mathbb{P}(MINE\_SET)$
MinePump_inv3 : $bell \in MINE \rightarrow BOOL$
MinePump_inv4 : $methane \in MINE \rightarrow BOOL$
MinePump_inv7 : $dom(bell) = MINE$
MinePump_inv8 : $dom(methane) = MINE$
**EVENTS**
**Initialisation**

**begin**
act1 : $MINE := \varnothing$
act3 : $bell := \varnothing$
act4 : $methane := \varnothing$
**end**
**Event** *updateBell* $\widehat{=}$
**any**
$m$
**where**
grd1 : $m \in MINE$
**then**
act1 : $bell :\mid methane(m) = TRUE \Rightarrow bell'(m) = TRUE$
**end**
**Event** *updateMethane* $\widehat{=}$
External event, DO NOT REFINE
**any**
$m$
$status$
**where**
grd1 : $m \in MINE$
grd2 : $status \in BOOL$
**then**
act1 : $methane(m) := status$
**end**
**Event** *addMine* $\widehat{=}$
External event, DO NOT REFINE
**when**
grd1 : $MINE = \varnothing$
**then**
act1 : $MINE := \{M\}$
act3 : $bell(M) := FALSE$
act4 : $methane(M) := FALSE$
**end**
**END**

Listing A.6: Mine pump example: MethaneSensor machine

**MACHINE** MethaneSensor
**SEES** MineContext
**VARIABLES**
methane Shared variable, DO NOT REFINE
MINE Shared variable, DO NOT REFINE
**INVARIANTS**
typing_methane : $methane \in \mathbb{P}(MINE\_SET \times BOOL)$
typing_MINE : $MINE \in \mathbb{P}(MINE\_SET)$
MinePump_inv1 : $MINE \in \mathbb{P}(MINE\_SET)$
MinePump_inv4 : $methane \in MINE \rightarrow BOOL$
MinePump_inv8 : $dom(methane) = MINE$
**EVENTS**
**Initialisation**
**begin**
act1 : $MINE := \varnothing$
act4 : $methane := \varnothing$
**end**
**Event** *updateMethane* $\widehat{=}$
**any**
$m$
$status$
**where**
grd1 : $m \in MINE$

```
        grd2 : status ∈ BOOL
    then
        act1 : methane(m) := status
    end
Event addMine ≙
    External event, DO NOT REFINE
    when
        grd1 : MINE = ∅
    then
        act1 : MINE := {M}
        act4 : methane(M) := FALSE
    end
END
```

Listing A.7: Mine pump example: PumpController_refinement machine

```
MACHINE PumpController_refinement
REFINES PumpController
SEES MineContext
VARIABLES
    methane     Shared variable, DO NOT REFINE
    waterLevel     Shared variable, DO NOT REFINE
    pump     Shared variable, DO NOT REFINE
    MINE     Shared variable, DO NOT REFINE
EVENTS
Initialisation
    extended
    begin
        act1 : MINE := ∅
        act2 : pump := ∅
        act4 : methane := ∅
        act5 : waterLevel := ∅
    end
Event high_water_detected ≙
    Internal Event
refines updatePump
    any
        m
    where
        grd2 : m ∈ MINE
        grd1 : waterLevel(m) = HIGH
        grd3 : methane(m) = FALSE
        grd4 : pump(m) = OFF
    then
        act1 : pump(m) := ON
    end
Event low_water_detected ≙
    Internal Event
refines updatePump
    any
        m
    where
        grd1 : m ∈ MINE
        grd2 : waterLevel(m) = LOW
        grd3 : pump(m) = ON
    then
        act1 : pump(m) := OFF
    end
```

**Event** *methane_detected* $\widehat{=}$
Internal Event
**refines** *updatePump*
**any**
$m$
**where**
grd1 : $m \in MINE$
grd3 : $pump(m) = ON$
grd4 : $methane(m) = TRUE$
**then**
act1 : $pump(m) := OFF$
**end**
**Event** *updateMethane* $\widehat{=}$
External event, DO NOT REFINE
**extends** *updateMethane*
**any**
m
status
**where**
grd1 : $\mathtt{m} \in \mathtt{MINE}$
grd2 : $\mathtt{status} \in \mathtt{BOOL}$
**then**
act1 : $\mathtt{methane(m)} := \mathtt{status}$
**end**
**Event** *updateWaterLevel* $\widehat{=}$
External event, DO NOT REFINE
**extends** *updateWaterLevel*
**any**
m
**where**
grd1 : $\mathtt{m} \in \mathtt{MINE}$
grd2 : $\mathtt{waterLevel(m)} = \mathtt{LOW} \lor \mathtt{waterLevel(m)} = \mathtt{MEDIUM} \lor \mathtt{waterLevel(m)} = \mathtt{HIGH}$
**then**
act1 : $\mathtt{waterLevel} :| (\mathtt{waterLevel(m)} = \mathtt{HIGH} \land \mathtt{waterLevel}' = (\mathtt{waterLevel} \lhd\!\!- \{\mathtt{m} \mapsto \mathtt{MEDIUM}\})) \lor (\mathtt{waterLevel(m)} = \mathtt{MEDIUM} \land \mathtt{waterLevel}' = (\mathtt{waterLevel} \lhd\!\!- \{\mathtt{m} \mapsto \mathtt{LOW}\})) \lor (\mathtt{waterLevel(m)} = \mathtt{MEDIUM} \land \mathtt{waterLevel}' = (\mathtt{waterLevel} \lhd\!\!- \{\mathtt{m} \mapsto \mathtt{HIGH}\})) \lor (\mathtt{waterLevel(m)} = \mathtt{LOW} \land \mathtt{waterLevel}' = (\mathtt{waterLevel} \lhd\!\!- \{\mathtt{m} \mapsto \mathtt{MEDIUM}\}))$
**end**
**Event** *addMine* $\widehat{=}$
External event, DO NOT REFINE
**extends** *addMine*
**when**
grd1 : $\mathtt{MINE} = \varnothing$
**then**
act1 : $\mathtt{MINE} := \{\mathtt{M}\}$
act2 : $\mathtt{pump(M)} := \mathtt{OFF}$
act4 : $\mathtt{methane(M)} := \mathtt{FALSE}$
act5 : $\mathtt{waterLevel(M)} := \mathtt{LOW}$
**end**
**END**

Listing A.8: Mine pump example: WaterLevelSensor_refinement machine

**MACHINE** WaterLevelSensor_refinement
**REFINES** WaterLevelSensor
**SEES** MineContext
**VARIABLES**
`waterLevel` Shared variable, DO NOT REFINE
`MINE` Shared variable, DO NOT REFINE
**EVENTS**
**Initialisation**
*extended*
**begin**
act1 : $\texttt{MINE} := \varnothing$
act5 : $\texttt{waterLevel} := \varnothing$
**end**
**Event** *high_to_medium* $\widehat{=}$
Internal Event
**refines** *updateWaterLevel*
**any**
$m$
**where**
grd1 : $m \in MINE$
grd2 : $waterLevel(m) = HIGH$
**then**
act1 : $waterLevel(m) := MEDIUM$
**end**
**Event** *medium_to_low* $\widehat{=}$
Internal Event
**refines** *updateWaterLevel*
**any**
$m$
**where**
grd1 : $m \in MINE$
grd2 : $waterLevel(m) = MEDIUM$
**then**
act1 : $waterLevel(m) := LOW$
**end**
**Event** *low_to_medium* $\widehat{=}$
Internal Event
**refines** *updateWaterLevel*
**any**
$m$
**where**
grd1 : $m \in MINE$
grd2 : $waterLevel(m) = LOW$
**then**
act1 : $waterLevel(m) := MEDIUM$
**end**
**Event** *medium_to_high* $\widehat{=}$
Internal Event
**refines** *updateWaterLevel*
**any**
$m$
**where**
grd1 : $m \in MINE$
grd2 : $waterLevel(m) = MEDIUM$
**then**
act1 : $waterLevel(m) := HIGH$
**end**
**Event** *addMine* $\widehat{=}$
External event, DO NOT REFINE
**extends** *addMine*
**when**
grd1 : $\texttt{MINE} = \varnothing$
**then**
act1 : $\texttt{MINE} := \{\texttt{M}\}$

```
        act5 : waterLevel(M) := LOW
    end
END
```

Listing A.9: Mine pump example: AlarmController_refinement machine

```
MACHINE  AlarmConrtoller_refinement
REFINES  AlarmController
SEES  MineContext
VARIABLES
    bell     Shared variable, DO NOT REFINE
    methane     Shared variable, DO NOT REFINE
    MINE     Shared variable, DO NOT REFINE
EVENTS
Initialisation
    extended
    begin
        act1 : MINE := ∅
        act3 : bell := ∅
        act4 : methane := ∅
    end
Event  trigger_alarm ≙
    Internal Event
refines  updateBell
    any
        m
    where
        grd1 : m ∈ MINE
        grd2 : methane(m) = TRUE
        grd3 : bell(m) = FALSE
    with
        status : status = TRUE
    then
        act1 : bell(m) := TRUE
    end
Event  updateMethane ≙
    External event, DO NOT REFINE
extends  updateMethane
    any
        m
        status
    where
        grd1 : m ∈ MINE
        grd2 : status ∈ BOOL
    then
        act1 : methane(m) := status
    end
Event  addMine ≙
    External event, DO NOT REFINE
extends  addMine
    when
        grd1 : MINE = ∅
    then
        act1 : MINE := {M}
        act3 : bell(M) := FALSE
        act4 : methane(M) := FALSE
    end
END
```

Listing A.10: Mine pump example: MethaneSensor_refinement machine

**MACHINE** MethaneSensor_refinement
**REFINES** MethaneSensor
**SEES** MineContext
**VARIABLES**
    `methane` Shared variable, DO NOT REFINE
    `MINE` Shared variable, DO NOT REFINE
**EVENTS**
**Initialisation**
    *extended*
    **begin**
        act1 : `MINE` := $\varnothing$
        act4 : `methane` := $\varnothing$
    **end**
**Event** *methane_leak* $\widehat{=}$
    Internal Event
**refines** *updateMethane*
    **any**
        $m$
    **where**
        grd1 : $m \in MINE$
    **with**
        status : `status` = `TRUE`
    **then**
        act1 : $methane(m) := TRUE$
    **end**
**Event** *addMine* $\widehat{=}$
    External event, DO NOT REFINE
**extends** *addMine*
    **when**
        grd1 : `MINE` = $\varnothing$
    **then**
        act1 : `MINE` := {`M`}
        act4 : `methane`(`M`) := `FALSE`
    **end**
**END**

Listing A.11: Mine pump example: re-composed machine

**MACHINE** MinePumpReunification
**REFINES** MinePump
**SEES** MineContext
**VARIABLES**
    `MINE`
    `pump`
    `bell`
    `methane`
    `waterLevel`
**EVENTS**
**Initialisation**
    *extended*
    **begin**
        act1 : `MINE` := $\varnothing$
        act2 : `pump` := $\varnothing$
        act3 : `bell` := $\varnothing$
        act4 : `methane` := $\varnothing$
        act5 : `waterLevel` := $\varnothing$
    **end**

**Event** $addMine \widehat{=}$
**extends** $addMine$
**when**
grd1 : $\texttt{MINE} = \varnothing$
**then**
act1 : $\texttt{MINE} := \{\texttt{M}\}$
act2 : $\texttt{pump}(\texttt{M}) := \texttt{OFF}$
act3 : $\texttt{bell}(\texttt{M}) := \texttt{FALSE}$
act4 : $\texttt{methane}(\texttt{M}) := \texttt{FALSE}$
act5 : $\texttt{waterLevel}(\texttt{M}) := \texttt{LOW}$
**end**
**Event** $trigger\_alarm \widehat{=}$
Internal Event
**refines** $updateBell$
**any**
$m$
**where**
grd1 : $m \in MINE$
grd2 : $methane(m) = TRUE$
grd3 : $bell(m) = FALSE$
**with**
status : $\texttt{status} = \texttt{TRUE}$
**then**
act1 : $bell(m) := TRUE$
**end**
**Event** $methane\_leak \widehat{=}$
Internal Event
**refines** $updateMethane$
**any**
$m$
**where**
grd1 : $m \in MINE$
**with**
status : $\texttt{status} = \texttt{TRUE}$
**then**
act1 : $methane(m) := TRUE$
**end**
**Event** $high\_water\_detected \widehat{=}$
Internal Event
**refines** $updatePump$
**any**
$m$
**where**
grd2 : $m \in MINE$
grd1 : $waterLevel(m) = HIGH$
grd3 : $methane(m) = FALSE$
grd4 : $pump(m) = OFF$
**then**
act1 : $pump(m) := ON$
**end**
**Event** $low\_water\_detected \widehat{=}$
Internal Event
**refines** $updatePump$
**any**
$m$
**where**
grd1 : $m \in MINE$
grd2 : $waterLevel(m) = LOW$
grd3 : $pump(m) = ON$
**then**
act1 : $pump(m) := OFF$
**end**
**Event** $methane\_detected \widehat{=}$
Internal Event

**refines** *updatePump*
**any**
$m$
**where**
grd1 : $m \in MINE$
grd3 : $pump(m) = ON$
grd4 : $methane(m) = TRUE$
**then**
act1 : $pump(m) := OFF$
**end**
**Event** *high_to_medium* $\widehat{=}$
Internal Event
**refines** *updateWaterLevel*
**any**
$m$
**where**
grd1 : $m \in MINE$
grd2 : $waterLevel(m) = HIGH$
**then**
act1 : $waterLevel(m) := MEDIUM$
**end**
**Event** *medium_to_low* $\widehat{=}$
Internal Event
**refines** *updateWaterLevel*
**any**
$m$
**where**
grd1 : $m \in MINE$
grd2 : $waterLevel(m) = MEDIUM$
**then**
act1 : $waterLevel(m) := LOW$
**end**
**Event** *low_to_medium* $\widehat{=}$
Internal Event
**refines** *updateWaterLevel*
**any**
$m$
**where**
grd1 : $m \in MINE$
grd2 : $waterLevel(m) = LOW$
**then**
act1 : $waterLevel(m) := MEDIUM$
**end**
**Event** *medium_to_high* $\widehat{=}$
Internal Event
**refines** *updateWaterLevel*
**any**
$m$
**where**
grd1 : $m \in MINE$
grd2 : $waterLevel(m) = MEDIUM$
**then**
act1 : $waterLevel(m) := HIGH$
**end**
**END**

# Glossary

**Agent:** Active objects (=processors) performing operations to achieve goals. The software-to-be is an agent. The agents can also come from the environment of the software-to-be. Human agents belong to the environment.

**Agent machine:** In chapter 4, an agent machine is a machine decomposing the initial machine and liked to a KAOS agent.

**Association:** Object the definition of which relies on other objects linked by the association.

**Conflict:** A conflict between goals exists if, under some boundary condition, the goals cannot be realised all together.

**Context:** Static part of an Event-B model, used to parametrize the model.

**Domain property:** Descriptive assertion on objects in the environment of the software-to-be. Domain invariant or hypothesis. A domain invariant is a property that is true in each state of a given domain object, for instance a physical law. A hypothesis is a property on a domain object supposed to be true.

**Entity:** Autonomous object (the definition of which does not rely on other objects).

**Environment:** Part of the universe able to interact with the software-to-be.

**Event:** Instantaneous object (alive only in one state) triggering operations realised by agents.

**Event-B:** Formal method for system specification. It uses set theory as a modelling notation and presents systems at different abstraction levels using the notion of refinement. Mathematical proof are operated to verify consistency between refinement levels.

**Expectation:** Goat assigned to an agent in the environment.

**External event:** An external event in a State-Based decomposition is an event declared in a machine, to keep the value of a shared variable synchronized with the other machine.

**Goal:** Prescriptive assertion standing for an objective to meet by means of cooperating agents; prescribes a set of desired behaviours. Requirements and expectations are particular cases of goals.

**Input:** Relationship showing that an object is used by an operation.

**Internal event:** An internal event in a State-Based decomposition is an event declared in a machine, to update the value of a shared variable.

**IsA:** Relationship between an object and a generalisation of this object.

**KAOS:** Goal-oriented methodology used in a software requirements engineering process. KAOS stand for Knowledge Acquisition in autOmated specification or Keep All Objects Satisfied.

**Machine:** Active part of an Event-B model that may contains variables eventually modified by events with respect to the declared invariants.

**Model:** Abstract representation of a composite system. An Objectiver model represents a composite system by means of concepts having several types, mainly objects, desired or not desired properties (goals, obstacles), and behaviours (operations).

**Object:** Focus of interest in the composite system being modelled, the instances of which can be identified separately and the states of which may evolve. Agents, events and entities are particular case of objects.

**Obstacle:** Condition (other than a goal) the satisfaction of which may prevent some goals from being achieved; defines a set of undesired behaviours.

**Operation:** Specifies state transitions of objects which are input and/or output of operations. Operations are performed by agents.

**Operationalization:** Relationship linking a requirement to operations. Is valid when each execution of operation (possibly constrained to that intent) ensures the requirement to be fulfilled. Sets a connection between desired properties (goals) and behaviours (operations)

**Output:** Relationship showing that an object is produced by an operation or modified by it (if the object is also an inputof it).

**Refinement:** Relationship linking a goal to other goals named sub-goals. Each sub-goal contributes to the satisfaction of the refined goal. The

conjunction of all sub-goals must be a sufficient condition to guarantee the refined goal.

**Requirement:** In goal oriented requirements engineering, it is a goal assigned to an agent of the software-to-be.

**Responsibility:** Relationship between an agent and a requirement/expectation . Is valid when the responsibility for achieving the requirement/expectation is assigned to an agent.

**Shared variable:** A shared variable in a State-Based decomposition is a variable that will be distributed between different machines. It will be updated by an internal event in one machine that is part of the decomposition and will be kept synchronized by external events in all the other machines.

**Stakeholder:** A person or an organisation concern by a project, that may be consulted at a moment of this project.

**Update event:** In chapter 4, an update event is an event representing the update of an element coming from the KAOS object model.

# List of Figures

# List of Tables

# Listings

# Bibliography


[Abrial, 1996] Abrial, J.-R. (1996). *The B-book: assigning programs to meanings*. Cambridge Univ Press.

[Abrial, 2009] Abrial, J.-R. (2009). Event model decomposition. `http://deploy-eprints.ecs.soton.ac.uk/109/`.

[Abrial, 2010] Abrial, J.-R. (2010). *Modeling in Event-B: System and Software Engineering*. Cambridge University Press.

[Aziz et al., 2009] Aziz, B., Arenas, A., Bicarregui, J., Ponsard, C., and Massonet, P. (2009). From goal-oriented requirements to event-b specifications. In *First Nasa Formal Method Symposium*, pages 96–105.

[Aziz et al., 2008] Aziz, B., Arenas, A., Martinelli, F., Matteucci, I., and Mori, P. (2008). Controlling usage in business process workflows through fine-grained security policies. In *TrustBus*, pages 100–117.

[Ball, 2008] Ball, E. (2008). *An Incremental Process for the Development of Multi-agent Systems in Event-B*. PhD thesis, University of Southampton. `http://eprints.ecs.soton.ac.uk/16575/`.

[Bicarregui et al., 2008] Bicarregui, J., Arenas, A., Aziz, B., Massonet, P., and Ponsard, C. (2008). Towards modelling obligations in event-b. In *ABZ*, pages 181–194.

[Bidoit and Mosses, 2004] Bidoit, M. and Mosses, P. D. (2004). *Casl User Manual - Introduction to Using the Common Algebraic Specification Language*, volume 2900 of *Lecture Notes in Computer Science*. Springer. `http://www.informatik.uni-bremen.de/cofi/wiki/index.php/CASL_user_manual`.

[Bresciani et al., 2004] Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., and Mylopulos, J. (2004). Tropos: An agent-oriented software development methodology. *Journal of Autonomous Agents and Multi-Agent Systems*, 8:203–236.

[Budinsky et al., 2003] Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., and Grose, T. J. (2003). *Eclipse modeling framework*. Pearson Education.

[Butler, 2009] Butler, M. (2009). Decomposition structures for event-b. *Integrated Formal Methods iFM2009, Springer, LNCS*, 5423:20–38.

[CEDITI, 2003] CEDITI (2003). *A KAOS Tutorial.* `http://www.cediti.be/`.

[Chung et al., 2000] Chung, L., Nixon, B. A., Yu, E., and Mylopoulos, J. (2000). *Non-Functional Requirements in Software Engineering.* Kluwer Academic Publishers.

[Clarke and Wing, 1996] Clarke, E. M. and Wing, J. M. (1996). Formal methods: state of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643.

[Falampin et al., 2009] Falampin, J., Butler, M., and Fitzgerald, J. (2009). Deploy deliverable d16: D2.1 pilot deployment in transportation (wp2). `http://www.deploy-project.eu/html/deliverables.html`.

[Gervais et al., 2009] Gervais, F., Gnaho, C., Laleau, R., Matoussi, A., and Semmak, F. (2009). Tacos livrable l1.2 : Kaos extension with non-functional properties. `http://tacos.loria.fr/drupal/?q=node/74`. Projet TACOS : Trustworthy Assembling of Components: frOm requirements to Specification ANR-06-SETI-017 Janvier 2007 - D´ecembre 2009.

[Hoare, 1985] Hoare, C. A. R. (1985). *Communicating Sequential Processes.* Prentice-Hall. `http://www.usingcsp.com/`.

[ISO/IEC, 2002] ISO/IEC (2002). Information technology – z formal specification notation – syntax, type system and semantics. `http://standards.iso.org/ittf/PubliclyAvailableStandards/c021573_ISO_IEC_13568_2002(E).zip`.

[Jones, 1990] Jones, C. B. (1990). *Systematic software development using VDM, 2nd ed.* Prentice Hall. ISBN 0-13-880733-7.

[Jouault et al., 2008] Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. (2008). Atl: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39.

[Landtsheer, 2007a] Landtsheer, R. D. (2007a). Deriving event-based security policy from declarative security requirements.

[Landtsheer, 2007b] Landtsheer, R. D. (2007b). *Elaborating Complete and Consistent Requirements for Security-Critical Systems.* PhD thesis, Université Catholique de Louvain. `http://www.info.ucl.ac.be/~rdl/thesis/`.

[Landtsheer and Ponsard, 2010] Landtsheer, R. D. and Ponsard, C. (2010). Deriving event-based usage control policies from declarative security requirements models. SEC-MDA 2010.

[Lapouchnian, 2005] Lapouchnian, A. (2005). Goal-oriented requirements engineering: An overview of the current research. Technical report, University of Toronto.

[Lecomte, 2009] Lecomte, T. (2009). Deploy deliverable d2: D14.03 electronic newsletter. `http://www.deploy-project.eu/newsletter/deploy-newsletter-03.pdf`.

[Letier, 2001] Letier, E. (2001). *Reasoning about Agents in Goal-Oriented Requirements Engineering*. PhD thesis, Université Catholique de Louvain.

[Matoussi, 2009] Matoussi, A. (2009). Expressing kaos goal models with event-b. In *Proceedings of Formal Methods 2009 Doctoral Symposium*, pages 60–67, Eindhoven, The Netherlands.

[Matoussi et al., 2008] Matoussi, A., Gervais, F., and Laleau, R. (2008). A first attempt to express kaos refinement patterns with event b. In *Proc. of the Int. Conf. on ASM, B and Z (ABZ). Lecture Notes in Computer Science, Springer-Verlag*, pages 12–14. Springer.

[Matoussi et al., 2009] Matoussi, A., Laleau, R., and Petit, D. (2009). Bridging the gap between kaos requirements models and b specifications. Technical Report TR-LACL-2009-5, LACL (Laboratory of Algorithms, Complexity and Logic), University of Paris-Est (Paris 12).

[Monin and Hinchey, 2003] Monin, J.-F. and Hinchey, M. G. (2003). *Understanding formal methods*. Springer Verlag.

[Métayer et al., 2005] Métayer, C., Abrial, J.-R., and Voisin, L. (2005). Rodin deliverable 3.2: Event-b language. `http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf`. `http://rodin-b-sharp.sourceforge.net`.

[OMG, 2007] OMG (2007). Meta object facility (mof) 2.0 query/view/transformation specification. v 1.0.

[OMG, 2009a] OMG (2009a). Omg unified modeling language (omg uml), infrastructure, v2.2. v 2.2.

[OMG, 2009b] OMG (2009b). Omg unified modeling language (omg uml), superstructure, v2.2. v 2.2.

[Pascal and Silva, 2009] Pascal, C. and Silva, R. (2009). Event-b model decomposition: A-style vs. b-style.

[Respect-IT, ] Respect-IT. *Objectiver Metamodel*.

[Respect-IT, 2009] Respect-IT (2009). Objectiver. `http://www.objectiver.com/`. version 3.0.0.

[REVER, 901] REVER (v 9.0.1). Db-main. `http://www.db-main.be`.

[Robinson, 2009] Robinson, K. (2009). A concise summary of the event-b mathematical toolkit. `http://wiki.event-b.org/images/EventB-Summary.pdf`.

[RODIN, 2010] RODIN (2010). Rodin platform. `http://www.event-b.org/`. version 1.3.

[Schneider, 2001] Schneider, S. (2001). *The B-method: an introduction*. Palgrave Macmillan, Basingstoke.

[Snook and Butler, 2006] Snook, C. and Butler, M. (2006). Uml-b: Formal modeling and design aided by uml. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122.

[van Lamsweerde, 2000] van Lamsweerde, A. (2000). Formal specification: a roadmap. In *Proceedings of the conference on The future of Software engineering*, pages 147–159. ACM New York, NY, USA.

[van Lamsweerde, 2001] van Lamsweerde, A. (2001). Goal-oriented requirements engineering: A guided tour. In *RE*, page 249.

[van Lamsweerde, 2009] van Lamsweerde, A. (2009). *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley.

[yah Said et al., 2009] yah Said, M., Butler, M., and Snook, C. (2009). Language and tool support for class and state machine refinement in uml-b. In *FM2009 - 16th International Symposium on Formal Methods*, number LNCS 5, pages 579–595. Springer.